Politechnika Wrocławska
Wydział Informatyki i Telekomunikacji

Kierunek: Informatyka (**INF)**
Specjalność: Systemy Informatyki w Medycynie (**IMT)**

PRACA DYPLOMOWA INŻYNIERSKA

Turowa gra strategiczna oparta na kontrolowaniu terenu

Jędrzej Piątek

Opiekun pracy
dr inż. Piotr Sobolewski

Słowa kluczowe: Python, gra, sztuczna inteligencja

WROCŁAW 2022

Streszczenie

Praca polegała na stworzeniu gry strategicznej, opierającej się na kontrolowaniu terenu. Program został napisany w języku Python[1] przy użyciu biblioteki Pygame[2] . Wybór technologii podyktowany był możliwościami stosunkowo prostej wizualizacji a także wcześniejszymi doświadczeniami z nią związanymi. Głównymi elementami pracy było stworzenie funkcjonalnego w miarę uniwersalnego systemu gry, oraz sztucznej inteligencji na poziomie pozwalającym na rywalizację z graczem ludzkim. Sztuczna inteligencja podejmuje decyzje na podstawie aktualnego stanu planszy, oraz w oparciu o współczynniki definiujące jej zachowanie.

Abstract

Thesis is about creating strategic game, where main goal is to controll terrain. Application was written in Python and main used library was Pygame. Technology was chosen based on possibilities of simple visualisation and also previous experiences with it. Most important parts of work was to create functional, universal game system and artificial intelligance with capibilites to compete with human player. Artificial inteligence is making decisions based on current state of the board and coefficients that are defining it’s behaviour.

Spis treści

1 Wstęp i opis istniejących rozwiązań

Podstawowym zadaniem było stworzenie strategicznej gry turowej. Gry tego rodzaju charakteryzują się podziałem rozgrywki na w miarę krótkie okresy czasowe nazwane turami. Zazwyczaj w czasie trwania tury jeden gracz wykonuje swoje ruchy. Charakterystyka tych ruchów może mocno różnić się w zależności od gry, ale najczęściej sprowadza się do zarządzania posiadanymi zasobami. Kiedy gracz wyczerpie możliwości ruchu kończy swoją turę. Schemat powtarza się dla wszystkich uczestników rozgrywki. Jednymi z bardziej znanych gier opartych na tym schemacie są serie Sid Meier's Civilization oraz Heroes of Might and Magic.

Dla stworzonego już systemu gry, zaimplementowano sztuczną inteligencję. Z racji na ogrom potencjalnego drzewa przeszukiwań, komputer nie byłby wstanie podejmować swoich decyzji w oparciu o kalkulację przyszłych ruchów przeciwnika, więc musi to robić w pełni na podstawie aktualnego stanu planszy, bez konkretnego modelu predykcji. Zamiast tego stworzono kilka algorytmów definiujących zachowanie komputera zarówno na podstawie wewnętrznych ustawień, jak i aktualnego stanu planszy.

Typowy podział na warstwę prezentacji i warstwę zasobów w pewnym sensie nie jest do końca zachowany w ramach projektu, jednak mimo tego da się rozróżnić które elementy programu są za co odpowiedzialne.

W tym przypadku warstwa prezentacji to gra więc naturalnym rozwiązaniem jest znalezienie biblioteki do tworzenia gier. Dla samego języka Python istnieje wiele bibliotek i narzędzi do tworzenia gier takich jak Pyglet[3] czy Ren'Py[4]. Wychodząc poza zakres narzędzi do Pythona, można by skorzystać z silnika Unity, albo bibliotek do c++ takich jak SFML czy Raylib. Każde z tych rozwiązań ma trochę inną charakterystykę, a ostateczny wybór padł na Pygame ze względu na możliwość realizacji założeń przy minimalnym nakładzie środków. Wymagania co do prezentacji w ramach pracy nie stoją na wysokim technologicznie poziomie, więc wybór rozwiązania był tym także podyktowany. Innym ważnym elementem były wcześniejsze doświadczenie autora.

W przypadku warstwy zasobów zagadnienie istniejących rozwiązań jest trochę bardziej skomplikowane. Plansza gry jest przechowywana w formie grafu więc poprawnym zdaje się wykorzystanie bibliotek implementujących takowe rozwiązania. Przykładowymi bibliotekami realizującymi to są python-igraph[5], NetworkX[6], graph-tool[7]. Problem z integracją któregoś z tych rozwiązań wynika ze specyfiki struktury potrzebnej w programie. Ze względu na to, po pobieżnym zapoznaniu się z wymienionymi wyżej technologiami, zdecydowano się własną implementację reprezentacji grafu. Daje to większą swobodę zarówno w związku z samą strukturą jak i operacjami na niej przeprowadzanymi.

2 Opis problemu i narzędzi

W poniższym rozdziale przedstawione zostaną główne założenia jakimi kierowano się przy tworzeniu gry, oraz to jak gra wygląda w praktyce. Akcje opisywane w rozdziale są z perspektywy gracza ludzkiego i mogą różnić się w teorii od tego jak wygląda to dla gracza komputerowego.

2.1 Założenia ogólne

Głównym celem gry, jest zajęcie wszystkich pól na mapie, poprzez odpowiednie manewrowanie dostępnymi zasobami. Przy tworzeniu programu przyjęto kilka założeń, wyznaczających jego ogólny zarys. Po pierwsze rozgrywka odbywa się w systemie turowym. Sprowadza się to do tego, że w czasie swojej kolejki, gracz ma możliwość wykonania określonych ruchów, po czym kończy swoją turę i ruchy wykonują inni uczestnicy. Kolejnym założeniem było stworzenie mapy jako struktury grafowej. Wierzchołki grafu traktowane są jako teren kontrolowany (lub nie) przez któregoś z graczy. Od strony wizualnej wierzchołki

widoczne są jako kwadraty, o kolorze odpowiadającym jednemu z graczy. Krawędzie oznaczają łączność pomiędzy poszczególnymi wierzchołkami, a tym samym możliwość przesuwania jednostek między nimi. Wstępnie ze względu na estetykę, nie zdecydowano się na rysowanie krawędzi w formie kresek. Zamiast tego po kliknięciu na dowolny z wierzchołków podświetlą się wszyscy jego sąsiedzi. Dodatkowo na podstawowych mapach, sąsiedztwo powinno być dosyć intuicyjne dla użytkownika, gdyż w praktyce pokrywa się z tym jak węzły ułożone są na ekranie. Domyślne mapy mają strukturę opartą na trójkątach równobocznych.

Wchodząc bardziej w specyfikę samej rozgrywki, także można wyróżnić kilka założeń. Każdy z graczy zaczyna rozgrywkę kontrolując przynajmniej 1 pole. Domyślnie, na mapie dla 4 graczy, każdy umiejscowiony jest w innym rogu mapy, w celu osiągnięcia możliwie jak najbardziej sprawiedliwej rozgrywki. W czasie trwania głównej części swojej tury, gracz ma możliwość przesuwana jednostek po mapie. Przesunięcie jednostki polega na zaznaczeniu własnego pola, po czym kliknięciu na jedno z oznaczonych sąsiednich pól. Po tej akcji, wszystkie jednostki, poza jedną, zostaną przeniesione na nowe pole. Jeżeli docelowe pole posiada jednostki nie należące do osoby wykonującej ruch, wywiązuje się walka. System walki jest stosunkowy prosty, większa liczba jednostek wygrywa i ponosi straty w kwocie równej liczbie jednostek które przegrały, jeżeli obie strony mają równą liczbę jednostek, zwycięzca wybierany jest w drodze losowania. Kolejnym elementem rozgrywki, jest rozmieszczanie nowych jednostek. Domyślnie następuje ono po zakończeniu ruchu jednostkami. Zliczana jest liczba pól aktualnie posiadanych przez gracza. Na tej podstawie otrzymuje on określoną liczbę jednostek do rozmieszczenia. Jednostki rozmieszczane są pojedynczo poprzez kliknięcia na zaznaczone pole.

2.2 Opis wykorzystanych narzędzi

Język Python jest aktualnie jednym z najpopularniejszych języków programowania. Posiada wszechstronne możliwości wykorzystania co czyni go odpowiednim to wszelakich typów projektów, poczynając od tworzenia stron internetowych do, jak w przypadku tej pracy, tworzenia gier. Jedną z głównych zalet języka są rozbudowane biblioteki oferujące mnogość rozwiązań różnych problemów. Dodatkowo z racji na swoją popularność, większość napotkanych problemów jest już dobrze udokumentowanych w internecie wraz z gotowymi rozwiązaniami.

Głównym narzędziem wykorzystanym w tworzeniu aplikacji była biblioteka Pygame. Oparta jest ona na licencji LGPL. Udostępnia mechanizmy ułatwiające tworzenie gier w przystępnej formie. Narzędzie to pozwoliło w prosty sposób na stworzenie interfejsu graficznego gry przy wykorzystaniu spritów. Dla przykładu dzięki temu detekcja kliknięty obiektu przebiegała poprzez interakcję z elementem z biblioteki, zamiast tworzenia całego mechanizmu wyszukiwanie takiego obiektu.

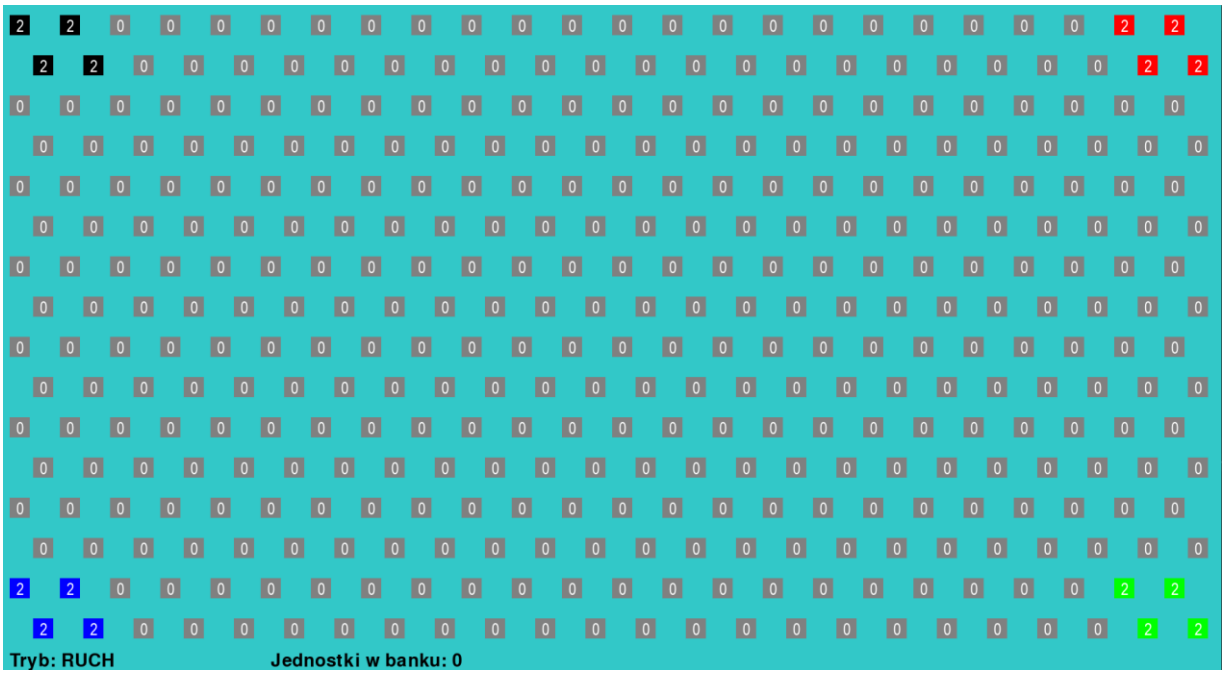
Do stworzenia pliku .exe umożliwiającego grę, wykorzystano bibliotekę PyInstaller[8] z nakładką graficzną auto-py-to-exe[9]. Pozwoliło to na wyeksportowanie plików ze wszystkim zależnościami i uruchamianie programu bez konieczności posiadania pythona lub potrzebnych bibliotek na komputerze użytkownika.

Z racji na naturę pracy większość rzeczy była pisana bez wykorzystania dodatkowych narzędzi, korzystając z w miarę normalnych elementów języka Python. Ciężko byłoby znaleźć narzędzia ułatwiające stworzenie podobnego projektu, nie będące jednocześnie prawie skończonym projektem. Dlatego też większość algorytmów została stworzona i poprawiana na bieżąco przez autora na podstawie jego doświadczeń i przemyśleń, co nie oznacza oczywiście, że podobne rozwiązania nie mogły być gdzieś indziej kiedyś stosowane.

2.3 Interfejs Graficzny

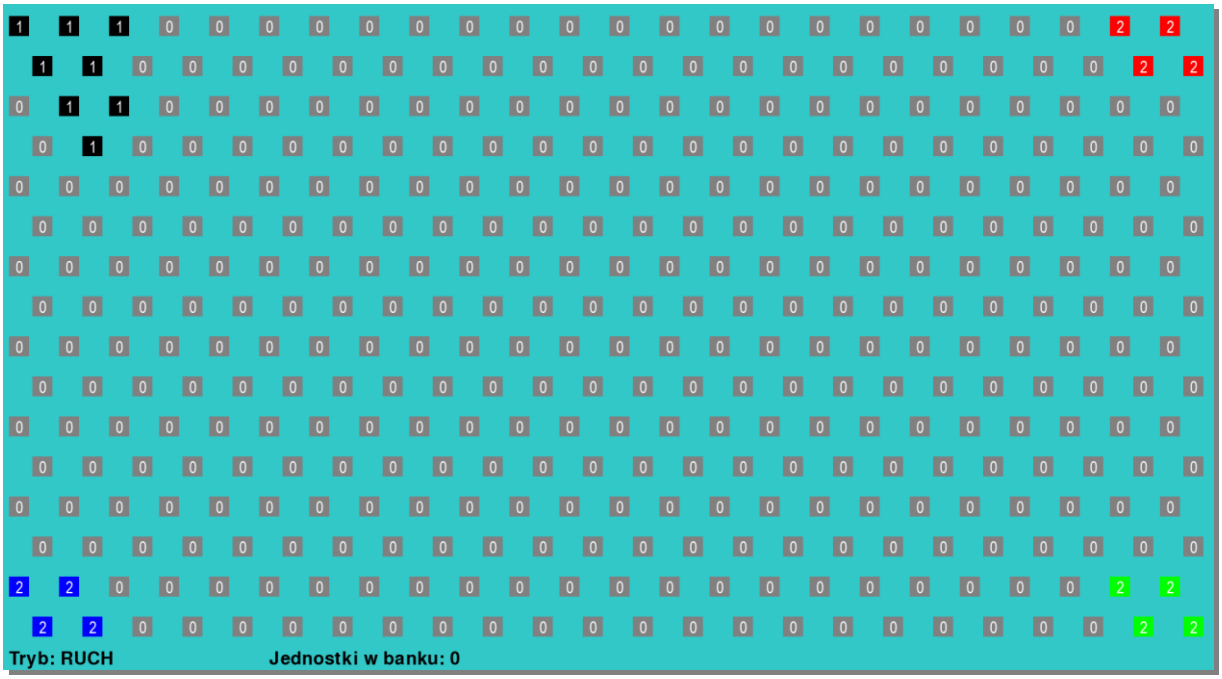
Z racji na chęć zachowania elastyczności i uniwersalności, grafika w programie jest dosyć prosta i surowa. Zmiana mapy na coś wyglądającego ciekawiej (na przykład nałożenie

mapy świata, albo czegoś podobnego), byłaby możliwa, jednak nie dałoby się tego sensownie przenieść na ogólny przypadek, więc nie próbowano tego wprowadzać. Interfejs zostanie omówiony poniżej na podstawie obrazków z gry.



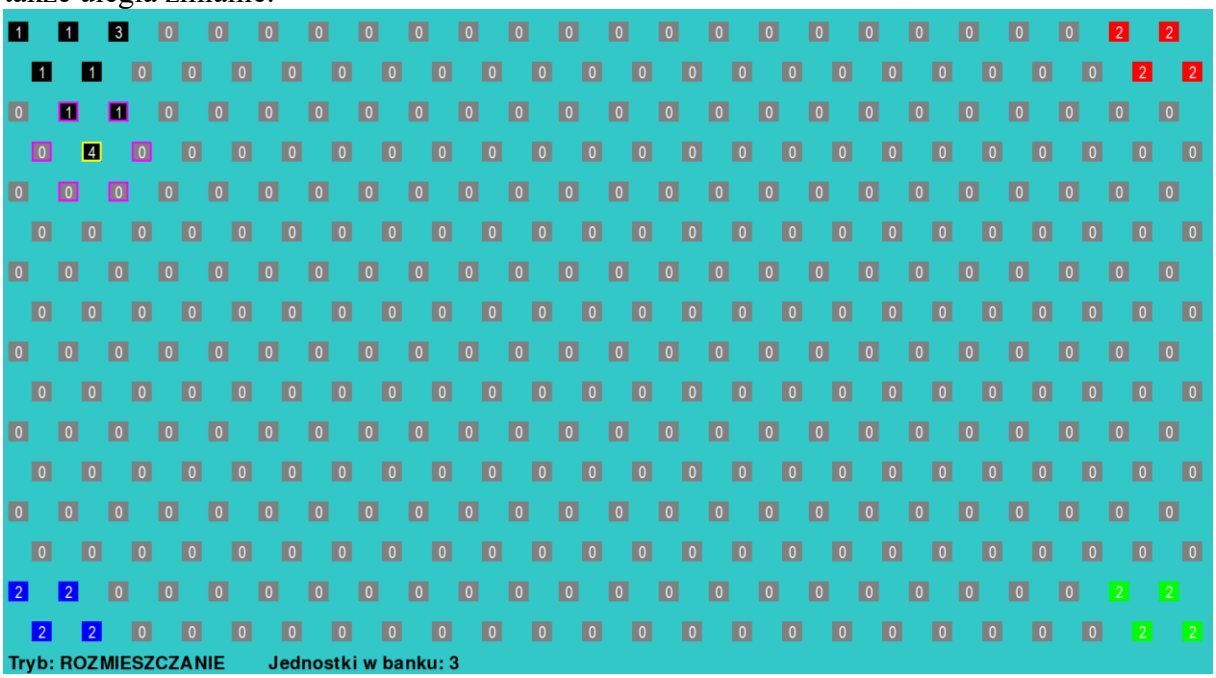
Rysunek 1: Ekran startowy dla mapy 16x24 (opracowanie własne)

Na powyższym rysunku (Rysunek 1) widać sytuację startową dla planszy 16x24. W każdym rogu znajduje się inny z graczy, co można poznać po różnych kolorach. Pomiędzy graczami, znajdują się pola neutralne, oznaczone na szaro. Liczba narysowana na każdym polu oznacza liczbę jednostek tam obecnych. Domyślnie każdy z graczy zaczyna z 2 jednostkami na swoich polach, a wierzchołki neutralne są puste. Na dole widzimy informacje o tym w jakiej fazie tury teraz się znajdujemy, oraz ile jednostek w banku posiadamy.



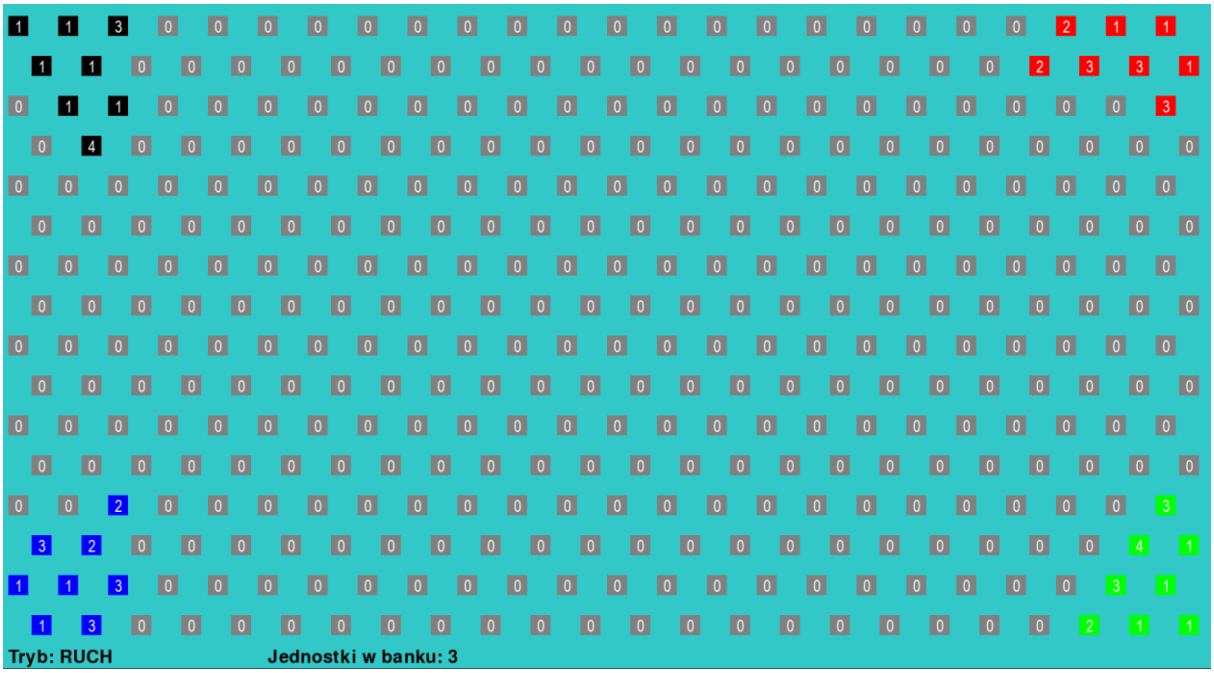
Rysunek 2: Po wykonaniu ruchów (opracowanie własne)

Teraz widać stan po wykonaniu kilku ruchów w pierwszej turze (Rysunek 2). Jak można zauważyć, pola przejęte przez gracza czarnego zmieniły kolor na czarny, liczba jednostek także uległa zmianie.



Rysunek 3: Tryb rozmieszczania jednostek (opracowanie własne)

Po kliknięciu spacji, tura przeszła w tryb rozmieszczania jednostek (Rysunek 3). Do banku jednostek dodana została liczba równa kontrolowanym polom (niektóre zostały już rozmieszczone). Na rysunku widać także, jak prezentuje się zaznaczone pole. Posiada ono żółtą obwódkę. Dookoła zaznaczonego pola oznaczono jego sąsiadów, poprzez fioletową obwódkę.



Rysunek 4: Początek 2 tury(opracowanie własne)

Po kolejnym naciśnięciu spacji, 3 graczy kontrolowanych przez komputer wykonało swój ruch i powrócono do gracza 1. Jak widać znowu tura jest w fazie ruchu. (Rysunek 4)

3 Implementacja

W poniższym rozdziale przedstawiony zostanie opis implementacji zarówno samej gry jak i sztucznej inteligencji.

3.1 Szczegóły mechanik gry

Kod omawiany w tym rozdziale nie jest za każdym razem dosłownie brany z programu, najczęściej od razu tłumaczono niektóre funkcje i pola klas, dla lepszej czytelności.

3.1.1 Klasy w programie

W ramach projektu zdefiniowano 3 główne klasy służące do implementacji systemu którego założenia zostały przedstawione we wcześniejszym rozdziale. Poniżej zostaną one szczegółowo omówione:

Node – Klasa ma dwojaką funkcjonalność. Każdy z obiektów tej klasy reprezentuje jeden z wierzchołków grafu, czyli jedno pole w rozgrywce. Tym samym przechowuje zarówno informacje potrzebne do funkcjonowania rozgrywki, jak i te potrzebne do wyświetlania wierzchołków. Pola klasy potrzebne wyłącznie do rozgrywki są następujące:

- ID – identyfikator pola
- Liczba jednostek – ile jednostek znajduje się w danym wierzchołku
- Właściciel pola – który z graczy kontroluje to pole, może być bez właściciela
- Wskaźnik zewnętrzności – pole którego wykorzystanie go zostanie omówione szczegółowo w dziale o sztucznej inteligencji, znajduje się tu dla wygody

Jeżeli chodzi o drugą kategorię pól, znajdują się tam następujące dane:

- Koordynaty X i Y – jako osobne pole, ale spełniają jedną funkcję, czyli umiejscowienia danego wierzchołka na planszy
- Zaznaczenie – Zmienna określająca czy wierzchołek został zaznaczony (kliknięty), jeżeli tak się stało, do tego wierzchołka dodana zostanie obwódka w innym kolorze, pozwalająca na zidentyfikowanie, który wierzchołek jest aktualnie wybrany
- Zaznaczenie jako sąsiada – Jeżeli sąsiedni wierzchołek został wybrany i jest zaznaczony, to ten wierzchołek zostanie zaznaczony jako jego sąsiad poprzez inną obwódkę. Pozwala to na identyfikację możliwych ruchów do wykonania.

GameTree – Klasa odpowiedzialna za drzewo spinające grafu. Jest to swego rodzaju lista sąsiedztwa, jednak jej elementami nie są obiekty klasy Node, ale liczby oznaczające ich ID. W aktualnym stanie projektu konstruktor klasy dostaje na wejściu wymiary mapy i na ich podstawie tworzy graf. Konstrukcja grafu jest następująca: Każdy wierzchołek nie będący wierzchołkiem granicznym, ma 6 sąsiadów (po lewej na górze, po lewej na tym samym poziomie, po lewej na dole i tak samo adekwatnie po prawej stronie). Wierzchołki graniczne nie będą posiadały któregoś z wyżej wymienionych sąsiadów, najmniej, bo 2 sąsiadów mogą mieć wierzchołki narożne. Tak skonstruowany graf stanowi dobrą bazę do ewentualnych przekształceń mapy. Jeżeli usunięto by któryś wierzchołków wystarczy usunąć jego id ze wszystkich list w obiekcie. Wydaje się to być dosyć uniwersalnym i prostym rozwiązaniem zarówno problemu tworzenia takiego grafu jak i wyświetlania go później na ekranie.

Player – Klasa ta jak sama nazwa wskazuje odpowiedzialna jest za przechowywanie danych gracza. Posiada następujące pola:

- ID – identyfikator gracza
- Typ gracza - czy jest to gracz ludzki czy komputerowy
- Bank jednostek – ile jednostek do rozstawienia gracz posiada w banku
- Nazwa gracza
- Liczba kontrolowanych pól – wyliczane na podstawie wszystkich pól, przechowywane jako pole dla sensownego dostępu

Dodatkowo klasa posiada pola używane jako współczynniki określające zachowanie AI, zostaną one szczegółowo omówione w rozdziale im poświęconym, ale w skrócie są to:

- Kondensacja – współczynnik mówiący o zwarciu pól
- Centralizacja – współczynnik mówiący o tym czy pola mają być bandowe czy centralne
- Poziom ataku – współczynnik określający barierę punktową poniżej której ruch nie jest warty wykonywania

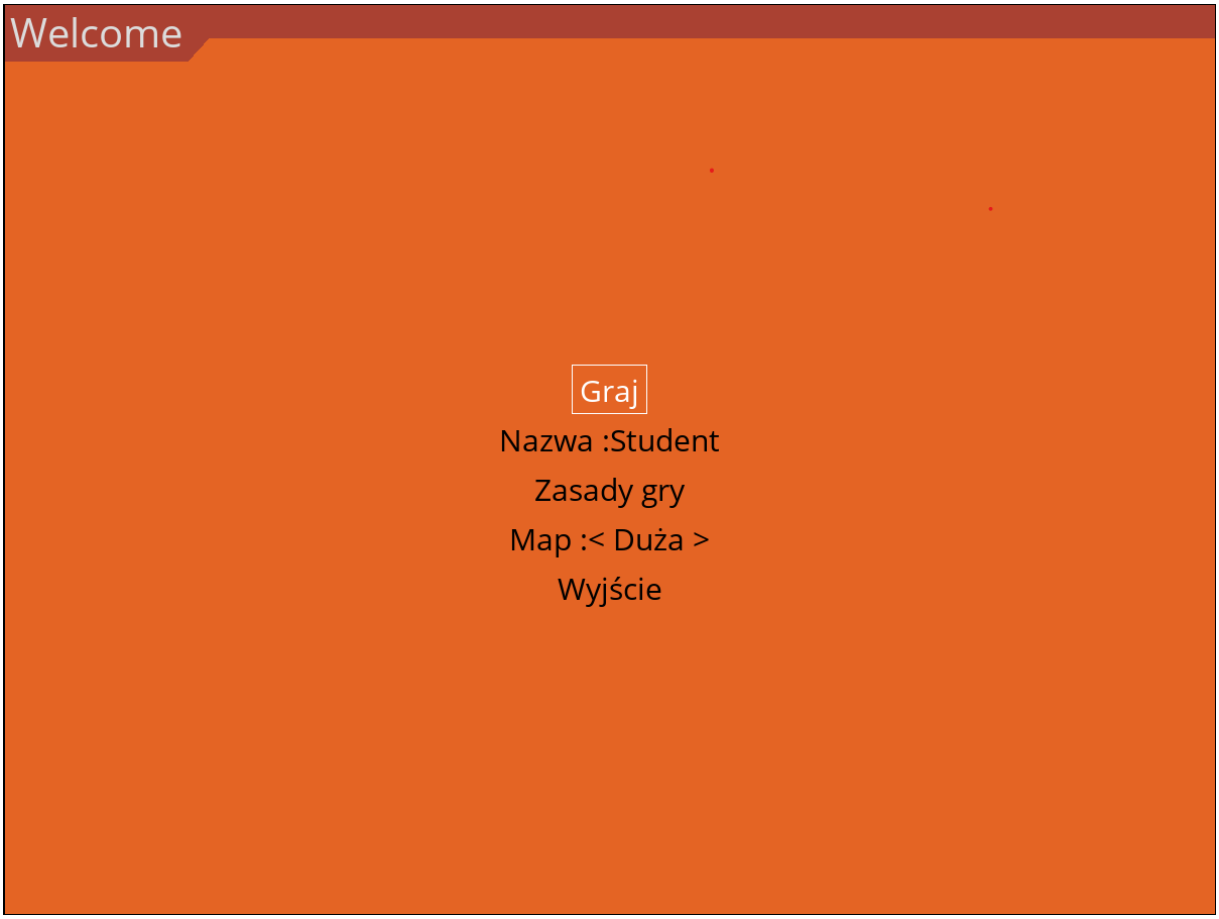
3.1.2 Rozgrywka

Poniżej omówione zostaną najważniejsze funkcje w programie. Pomijając ustawianie zmiennych koniecznych do utworzenia okna, program zaczyna się od wczytania mapy, czyli utworzenia wszystkich obiektów klasy Node oraz jednego obiektu klasy GameTree będącego grafem reprezentującym planszę. Następnie tworzeni są gracze, a graczom komputerowym przypisywanie są współczynniki służące do wykonywania przez nich ruchów. Wszyscy gracze zapisywani są kolejno na liście graczy. W tym momencie zaczyna się pętla główna programu. Jeżeli aktualnie wykonującym ruchem jest gracz komputerowym, to wywoływana jest funkcja odpowiedzialna za wykonanie przez komputer swojego ruchu, po wyjściu z tej funkcji następuje zmiana gracza. W przypadku kiedy ruch wykonuje gracz ludzki, program oczekuje na eventy takie jak kliknięcie myszką, lub wciśnięcie spacji. Po kliknięciu myszką na któryś z wierzchołków zostaje on zaznaczony. Jeżeli zaznaczony wierzchołek należy do gracza aktualnie wykonującego ruch i klikniemy na sąsiadujące pola, wywoływana jest funkcja odpowiedzialna za przemieszczanie jednostek. Na wejściu otrzymuje ona 2 obiekty typu Node, czyli początkowe i końcowe pole, które zostały zaznaczone. Następnie na podstawie jednostek obecnych w obu wierzchołkach wylicza rezultat walki i przypisuje nowe wartości do każdego z pól. Po tej akcji, niezależnie od rezultatu pole docelowe zostaje zaznaczone (kliknięte). Dotyczy to zarówno ruchu z własnego pola na własne, jak i z własnego pola na cudze. W przypadku przemieszczania jednostek pomiędzy własnymi polami, finalna liczba jednostek na docelowym polu nie może przekroczyć 12 (wartość subiektywnie ustalona przez twórcę, bez większego znaczenia), jeśli liczba ta przekraczałaby 12, to ruch nie jest wykonywany. Jeżeli gracz znajduje się w fazie ruchu i kliknie spację, przejdzie do fazy rozmieszczania jednostek. Na początku wywołana zostanie funkcja zliczająca wszystkie kontrolowane przez nas pola. Jest to tak naprawdę metoda klasy Player, która przyjmuje na wejściu listę wszystkich wierzchołków, i iterując po nich sprawdza które należą do danego gracza. Następnie zapisuje otrzymaną wartość w polu klasy do tego przeznaczonym. W tej fazie można rozmieścić posiadane w banku jednostki. Jednostki rozmieszcza się pojedynczo poprzez kliknięcie na zaznaczone już pole. Po kliknięciu spacji w tej fazie, wywołuje się funkcja kończąca tura, która zwiększa iterator listy z graczami (wartość ta jest modulo liczba graczy, więc jeżeli przekroczy pewną wartość to spada do 0).

3.1.3 Dodatki

W ramach projektu zostało utworzonych kilka dodatkowych funkcjonalności, które nie są niezbędne do działania głównego mechanizmu, są jednak przydatne.

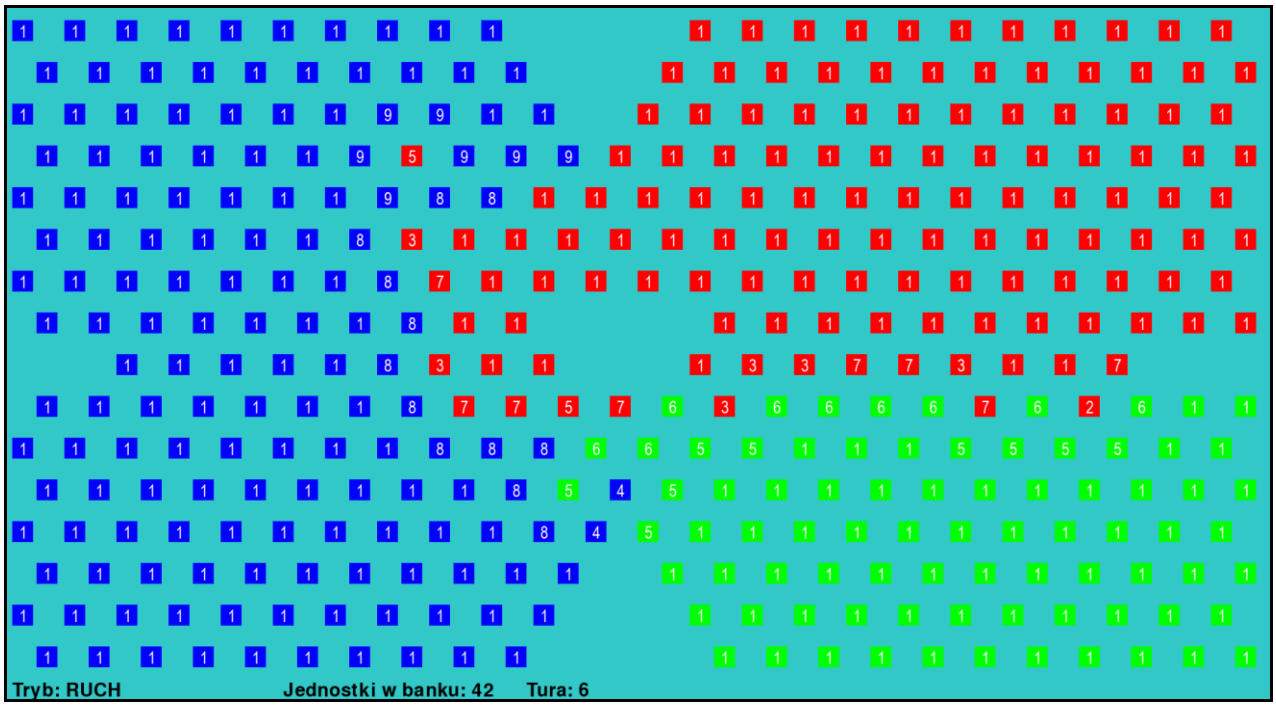
Wczytywanie mapy z pliku: Jak nazwa wskazuje funkcja służy do wczytania mapy gry z pliku. Plik powinien być plikiem .txt i mieć następujący format: 1 linia to liczba wierszy, 2 linia to liczba kolumn, kolejne linie to linie z wierzchołkami, każda zawiera 4 informacje oddzielone tabulatorami: Id wierzchołka, koordynat x, koordynat y, Id gracza kontrolującego wierzchołek. Funkcja na podstawie uzyskanych danych tworzy kolejne obiekty typu Node, oraz obiekt GameTree. Obiekt typu GameTree jest ostatecznie zwracany.
Menu Główne – Do programu dodano proste menu z możliwością wyboru wielkości mapy i nazwy gracza. Menu zostało stworzone przy pomocy biblioteki pygame-menu[10]. Domyślnie wielkość mapy ustawiona jest na dużą (24x16), alternatywnie można wybrać małą (12x8) lub wariację dużej z usuniętymi niektórymi wierzchołkami. Poniżej aktualny wygląd menu (Rysunek 5).



Rysunek 5: Menu główne (opracowanie własne)

Skróty klawiszowe – Stworzono skróty klawiszowe umożliwiające dodawanie jednostek do wierzchołka poprzez kliknięcie klawisza ‘a’ oraz dodania maksymalnej liczby jednostek do wybranego wierzchołka poprzez kliknięcie klawisza ‘m’.

Edycja mapy – Rozszerzenie wczytywania z pliku do usuwania niektórych wierzchołków z grafu zmieniając tym samym mapę. Po ostatniej linii oznaczającej wierzchołek, kolejne linie oznaczają wierzchołki które należy usunąć. Poniżej ilustracja trwającej rozgrywki na mapie z której usunięto kilka wierzchołków. (Rysunek 6)



Rysunek 6: Mapa bez niektórych wierzchołków (opracowanie własne)

3.2 Sztuczna inteligencja

W rozdziale tym omówione zostaną aspekty związane ze sztuczną inteligencją obecną w programie. Przedstawione założenia, którymi kierowo się przy jej tworzeniu i ich ostateczny efekt.

3.2.1 Założenia efektywności, złożoności i działania

Podstawowym założeniem którym kierowano przy tworzeniu AI, było zrobienie systemu na tyle efektywnego by pozwalał na rywalizowanie z graczem ludzkim a tym samym tworzenie ciekawych rozgrywek. Dodatkowo, algorytm zwracający ruchy dla komputera nie może być zbyt złożony obliczeniowo, żeby dla dużych instancji problemu gra zachowywała płynność rozgrywki. Podstawowy model sztucznej inteligencji został utworzony na podstawie kilku współczynników, z których każdy oznacza inne tendencje komputera. Podstawowymi wskaźnikami są kondensacja, centralizacja i poziom ataku. Są one ustawiane jako pola klasy Player. Na ich podstawie komputer w momencie wykonywania ruchu dokonuje ewaluacji wszystkich możliwych ruchów ofensywnych, tym samym każdy legalny ruch ma przypisaną jakąś wartość. Na tej podstawie komputer wybiera najlepszy ruch w danym momencie, jeżeli jego wartość jest większa od współczynnika poziomu ataku ruch zostaje wykonany. Po wykonaniu ruchu, gdzie ruch to nie tura, tylko wykonanie jednej akcji przemieszczenia jednostek między wierzchołkami, komputer ponownie dokonuje ewaluacji dla nowego stanu planszy. Jeżeli okazuje się, że żaden ruch nie został wykonany komputer sprawdza, czy jest w stanie wykonać ruchy pomiędzy swoimi polami, w celu poprawy pozycji. Jako poprawę pozycji rozumie się przemieszczenie jednostek na zewnątrz swojego skupiska pól. W tym celu każdemu posiadanemu wierzchołkowi przypisuje wartość, oznaczającą to jak bardzo jest na zewnątrz (jedno z pól klasy Node). Następnie po przypisaniu tej wartości do każdego posiadanego pola następuje przemieszczanie jednostek między nimi. W tym wypadku nie trzeba przeprowadzać rewaluacji po każdym ruchu, gdyż stan posiadanych pól się nie zmienia, a tylko ich zawartość, która nie jest brana pod uwagę. Jeżeli ostatecznie żaden ruch nie został wykonany w całym przebiegu pętli, komputer przechodzi do fazy rozmieszczania jednostek. Tutaj ponownie wykorzystywany jest algorytm oceny bycia na zewnątrz danego

wierzchołka. Dla pól będących w pierwszej warstwie, kolejno przypisywane są jednostki z banku aż do ich wyczerpania.

3.2.2 Implementacja

Omówione zostanie tutaj wyliczanie ewaluacji na podstawie danych współczynników, oraz to jak dokładnie powinny one wpływać na zachowanie komputera. Współczynniki kondensacji i centralizacji, mogą mieć wartości od 1 do 100, gdzie 1 oznacza brak chęci centralizacji/kondensacji, a 100 maksymalną chęć. W takim układzie wartość 50 oznacza efektywnie ignorowanie danego współczynnika, gdyż dla dowolnego stanu planszy ewaluacja będzie zawsze taka sama. Współczynnik poziomu ataku w teorii może mieć wartości od 0 do 110 000, jednakże w praktyce są to wartości z zakresu od 0 do 5000. W poniższych przypadkach (poza pierwszym) mowa jest o ewaluacji ruchów ofensywnych (przemieszczeniu na nie swoje pole). Dla każdej funkcji, poniżej znajduje się listing z jej kodem.

Na początek omówiony zostanie algorytm przypisywania wierzchołkom wartości „bycia na zewnątrz". Oznacza to że najwyższy wynik będą miały punkty graniczące i wierzchołkami nie należącymi do danego gracza, a najniższy te znajdujące się najdalej od granicy jego pól. Ocena dokonywana jest następująco. Każdy z wierzchołków zaczyna w wynikiem 0. Za każdy graniczący wierzchołek nie należący do gracza wartość zwiększa się o 1. Tak utworzona zostaje warstwa zewnętrzna. Następnie dla każdego z pól które mają wartość 0 przypisywana jest wartość równa średniej ich sąsiadów podzielonej przez 6. Żeby nie zaburzyć wyników, wartości dodawane są dopiero po całkowitym przeiterowaniu po wierzchołkach (chodzi o to, że nowy wynik od razu byłby brany pod uwagę przy wyliczaniu kolejnych wartości, co byłoby nie poprawne). Proces powtarzany jest aż każdy z wierzchołków będzie miał przypisaną jakąś niezerową wartość. Wyniki z tego wykorzystywane są przez komputer do przemieszczanie jednostek na zewnątrz swoich pól. Poniżej kod wyliczający te wartości (Listing 1).

Listing 1: Ustawianie wartości bycia na zewnątrz dla wierzchołków (opracowanie własne)

```
def set_nodes_outside_score(player, owned_nodes, neighbours_list,
full_neighbour_list):
    nodes_number = player.controlled_nodes
    # wyzerowanie wartosci w nodach, takie rozwiazanie wydaje sie najprostrze
    for node in owned_nodes:
        node.outside_score = 0
    for i in range(nodes_number):
        val = len(neighbours_list[i])  # przypisanie wartosci
        owned_nodes[i].outside_score = val
    run = True
    while run:
        # nie chce odrazu zmieniac pola, bo wplyneloby to na inne wyniki w tej
"warstwie", wiec dodaje po przejsciu calej warstwy
        results_list = []
        id_list = []
        for i in range(nodes_number):
            if owned_nodes[i].outside_score == 0:
                score_sum = 0
                non_zeros = 0
                for node in full_neighbour_list[i]:
                    score_sum += node.outside_score
                    if node.outside_score > 0:
                        non_zeros += 1
                if non_zeros > 0:
                    results_list.append(
                        score_sum / (6 * non_zeros))  # jezeli nie jest w wartswie
zewnetrznej to bedzie <0
```

```
                id_list.append(owned_nodes[i].id)
        for node_id in id_list:
            for node in owned_nodes:
                if node.id == node_id:
                    res_id = id_list.index(node_id)
                    node.outside_score = results_list[res_id]
        # Sprawdz czy zostaly jakies zerowe
        run = False
        for node in owned_nodes:
            if node.outside_score == 0:
                run = True
                break
```

Wyliczanie kondensacji, polega na przypisaniu do każdego z dostępnych ruchów wartości zmiany skupienia punktów. Algorytm powinien działać na bazie poprzedniego omawianego algorytmu, jednak ze względu na to, że dla każdego ruchu trzeba byłoby od nowa wyliczać wartość dla każdego wierzchołka po czym je sumować, zdecydowano się na uproszczoną wersję z podobnym działaniem. Działa następująco, za każdy sąsiadujący wierzchołek należący do tego gracza, wartość punktowa spada o 1, za każdy obcy wierzchołek wzrasta o 1. Tym sposobem z racji na naturę mapy i zasad gry, możliwe wyniki mieszczą się w zakresie od -6 do 4. Następnie wyniki przekształcane są na zbiór od 0 do 100, gdzie -6 przechodzi na 100, a 4 przechodzi na 0. Następnie dla każdego wyniku uwzględniany jest współczynnik kondensacji i ostatecznie zwracana jest wartość od 1 do 101. Poniżej kod (Listing 2).

Listing 2: Wyliczanie kondensacji (opracowanie własne)

```
def evaluate_condensation(self,move_table,owned_nodes, game_tree, neighbours_list,
full_neighbours_list,all_nodes):
    eval_move_table = []
    scores = []
    for single_node_moves in move_table:
        for move in single_node_moves:  # move to obiekt klasy node
            neighbours_amount = len(game_tree.Adjacency_list[move.id])
            player_neighbours_amount = 0
            # policz ile ma sasiadow aktualnego gracza
            for node in all_nodes:
                if node.id in game_tree.Adjacency_list[move.id]:  # jezeli jest
sasiadem tego punktu
                    if node.owner_id == self.id:
                        player_neighbours_amount += 1
            # policz ile ma innych sasiadow
            other_neighbours_amount = neighbours_amount - player_neighbours_amount

            scores.append(other_neighbours_amount - player_neighbours_amount)  #
maksymalnie 4, minimalnie -6
    delta = 10
    # przejscie na skale 0 do 100, 100 to najbardziej skondensowany, czyli
najmniejszy wynik
    for score in scores:
        score += 6
        score = (delta - score) / delta * 100
        final_eval = ((self.condensation - 50) * (score - 50)) / 50 + 50
        eval_move_table.append(final_eval + 1)
    return eval_move_table
```

Wyliczanie centralizacji, to ocena tego jak bardzo centralny jest wierzchołek docelowy i przeliczenie tego na wartość końcową z uwzględnieniem współczynnika centralizacji. Początkowo wyliczany jest środek planszy (dla 16x24, będzie to 7.5 ; 11.5, gdyż indeksujemy od 0), następnie każdemu ruchowi przypisywana jest wartość na podstawie różnicy jego współrzędnych i współrzędnych centralnych (najmniej 1, dla najbardziej centralnych, najwięcej 19 dla punktów w rogach). Następnie wartości te przemieniane są na skalę od 0 do 100, gdzie centralne punkty są bliżej 100, a punkty bandowe bliżej 0. Na podstawie tego wyniku, współczynnika centralizacji, wyliczane są ostateczne zwracane wartości. Końcowe wartości są w zakresie od 1 do 101. Poniżej kod (Listing 3).

Listing 3: Wyliczanie centralizacji (opracowanie własne)

```
def evaluate_centralization(self, move_table, owned_nodes, game_tree):
    # maksymalny score dla miejsc nodow centralnych czyli dla 8x12 to bedzie (3,5)
(3,6), (4,5), (4,6)
    eval_move_table = []
    x_len = game_tree.columns
    y_len = game_tree.rows
    x_centre = float(x_len / 2) -0.5 # centralny punkt poziomo
    y_centre = float(y_len / 2)  -0.5 # centralny punkt pionowo
    max_length = float((x_len + y_len) / 2)  # maksymalna odleglosc od srodka
    for single_node_moves in move_table:
        for move in single_node_moves:  # move to obiekt klasy node
            move_cords = move.manhattan_pos(game_tree)
            score = manhattan_score([y_centre, x_centre], move_cords)  # i mniej
tym bardziej centralne
            score_adjusted = (max_length - score) / max_length * 100  # im wiecej
tym bardziej centralne, od 0 do 100
            final_eval = ((self.centralization - 50) * (score_adjusted - 50)) / 50
+ 50
            eval_move_table.append(final_eval+1)
    return eval_move_table
```

Ostatnim elementem wyliczanym przy ocenie ruchu, jest wartość ataku. Nie zależy ona od żadnego ze współczynników, co oznacza że w takiej samej sytuacji będzie identyczna dla każdego z graczy. Wylicza się ją na podstawie strat ponoszonych w trakcie wykonywania ruchu. W ten sposób, zajęcie pola bez strat daje wartość maksymalną, podczas gdy każda poniesiona strata obniża tą wartość. Wartość jest jeszcze mniejsza jeżeli, atak nie przynosi efektu w postaci zajęcia pola. Poniżej kod (Listing 4).

Listing 4: Wyliczanie wartości ataku (opracowanie własne)

```
def evaluate_attack(self, move_table, owned_nodes, game_tree):
    eval_move_table = []
    node_iterator = 0
    for single_node_moves in move_table:
        for move in single_node_moves:  # move to obiekt klasy node
            if owned_nodes[node_iterator].units_amount > 1:
                attackers = owned_nodes[node_iterator].units_amount
                if move.units_amount == 0:  # podboj bez strat
                    eval_move_table.append(1 * attackers)
                elif owned_nodes[node_iterator].units_amount > move.units_amount +
1:  # pewny podboj, ze stratami
                    # attackers = owned_nodes[node_iterator].units_amount -1
                    defenders = move.units_amount
                    # loss =  defenders
                    # score = 1 - 0.05* defenders
```

```
                    eval_move_table.append((1 - 0.05 * defenders) * attackers)
                elif owned_nodes[node_iterator].units_amount == move.units_amount
+ 1:  # 50/50
                    eval_move_table.append(0.5)
                else:  # mniej jednostek, czyli przegrana bitwa
                    defenders = move.units_amount
                    eval_move_table.append((0.75 - 0.05 * defenders) * attackers)
            else:
                eval_move_table.append(0)
        node_iterator += 1
    return eval_move_table
```

Na podstawie tak wyliczonych ocen dla każdego ruchu tworzona jest końcowa ewaluacja. Wybierany jest ruch z najlepszym wynikiem, jeżeli jest kilka najlepszych to pierwszy z listy. Jeśli wartość tego ruchu jest lepsza od wartości współczynnika poziomu ataku ruch zostaje wykonany, w przeciwnym wypadku nie dzieje się nic i algorytm przechodzi do części przesuwania własnych jednostek lub kończenia tury (szczegóły opisane w poprzednim podrozdziale). Widać z tego, że wysoka wartość współczynnika poziomu ataku, może powodować, że niektóre ruchy nie zostaną wykonane, co może być interpretowane jako pasywne zachowanie komputera. Poniżej fragment kodu pętli dla ruchu komputera w którym podejmuje on decyzje o następnym ruchu (Listing 5).:

Listing 5:Fragment kodu głównej pętli ruchu komputera (opracowanie własne)

```
# wyliczanie wartosci dla ruchu
condensation_table = player.evaluate_condensation(neighbours_list, owned_nodes,
game_tree, neighbours_list,
                                                  full_neighbours_list, all_nodes)
centralization_table = player.evaluate_centralization(neighbours_list,
owned_nodes, game_tree)
attacks_table = player.evaluate_attack(neighbours_list, owned_nodes, game_tree)
units_backup_table = []
for i in owned_nodes:
    units_backup_table.append(int(i.units_amount))
moves_number = len(centralization_table)
final_eval_table = np.zeros(moves_number)
for i in range(moves_number):
    final_score = condensation_table[i] * centralization_table[i] *
attacks_table[i]
    final_eval_table[i] = final_score
try:
    best_move = np.argmax(final_eval_table)

    if final_eval_table[best_move] > player.threshold:
        iterator = 0
        node_iter = 0
        for one_node_neighbours in neighbours_list:
            for node in one_node_neighbours:
                if iterator == best_move:
                    move_units(node, owned_nodes[node_iter])
                iterator += 1
            node_iter += 1
except ValueError:  # to jest koniec gry, juz nie ma mozliwych ruchow bo wszystko
zajal
    running = False
    break
```

4 Wyniki eksperymentów

W poniższym rozdziale zostaną przedstawione wyniki związane z eksperymentami przeprowadzonymi ze sztuczną inteligencją. Głównym celem eksperymentów było znalezienie najlepszej kombinacji współczynników. Ogólnie nie możliwe jest znalezienie doskonałej kombinacji, która zawsze wygra rozgrywkę. Wynika to z tego, że w grze bierze udział więcej niż 2 graczy, oraz wynik w danej grze oznacza w rzeczywistości tylko wynik przeciwko danemu zestawowi współczynników reprezentowanemu przez innych graczy. Wszystkie testy zostały przeprowadzone na mapie 24x16.

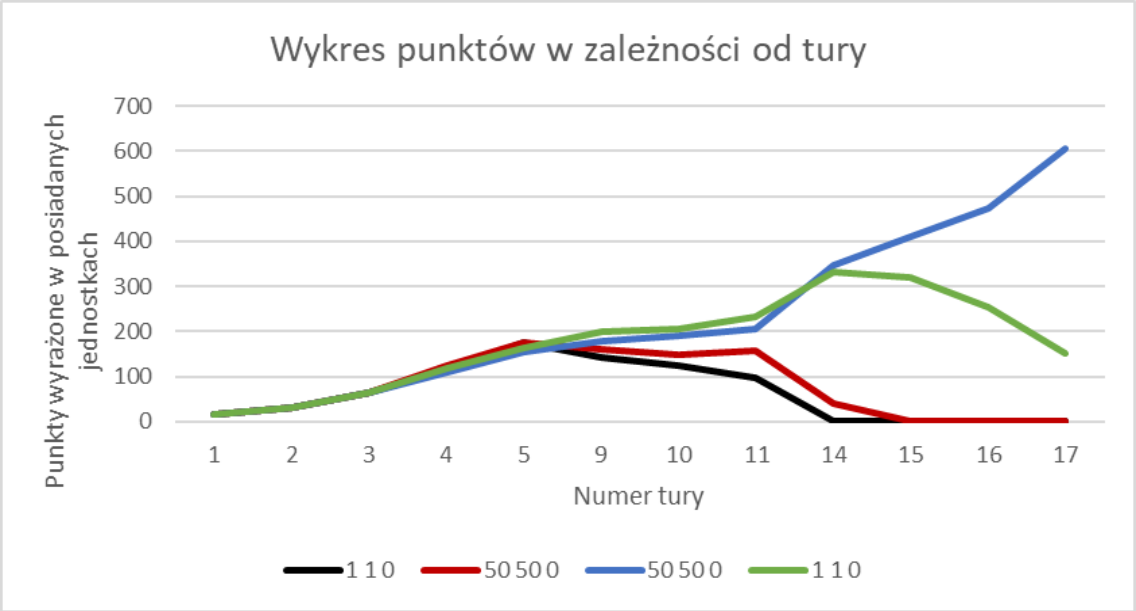
4.1 Porównanie efektywności współczynników sztucznej inteligencji

W ramach pierwszych testów gracze 2 i 3 mieli współczynniki kondensacji i centralizacji ustawione na 50. Oznacza to, że dowolnej ewaluacji pozycji funkcje odpowiadające tym współczynnikom zawsze zwracały 50, więc jedynym istotnym elementem była ewaluacja wartości ataku, która dokonywana jest na podstawie kalkulacji zysków i strat przy wykonywanym ruchu. Z racji, że większość ruchów w początkowej fazie będzie miało taką samą ewaluację i wybierany jest wtedy pierwszy ruch z listy, to gracze ci będą dążyć w lewy górny róg. Oznacza to że pozycja gracza w tym rogu, jest naturalnie gorsza. W ramach tej serii testów wartości współczynników zmieniano dla gracza 1 i 4. W tabelach nie uwzględniono wszystkich tur, zostawiono pierwsze 5 dla każdej serii po czym wybrano te w których zazwyczaj zachodziły większe zmiany. Poza rzeczami wspomnianymi powyżej, należy też pamiętać, że w grze występuje minimalnym element losowości (przy starciu takiej samej liczby jednostek), który jednak może mieć swój wpływ na wyniki.

Poniżej tabela (Tabela 1) i wykres (Rysunek 7) dla pierwszego zestawu, w tabeli na zielono zaznaczono zwycięzcę, na wykresie etykiety serii oznaczają współczynniki:

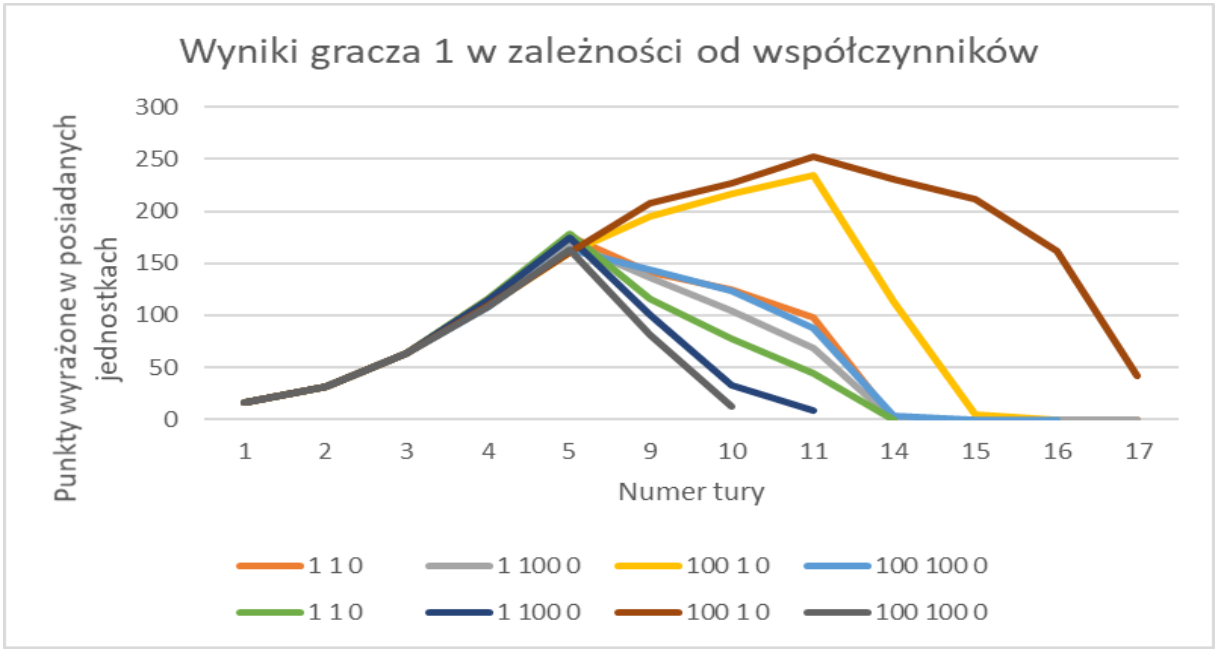
Tabela 1. Wyniki dla pierwszego zestawu danych (opracowanie własne)

gracz	kondensacja	centralizacja	chęć ataku	1	2	3	4	5	9	10	11	14	15	16	17
1	1	1	0	16	32	64	116	176	141	125	98	0	0	0	0
2	50	50	0	16	32	64	123	177	162	149	157	40	0	0	0
3	50	50	0	16	32	64	110	154	179	191	205	346	409	473	605
4	1	1	0	16	32	64	118	164	200	206	232	332	320	254	152

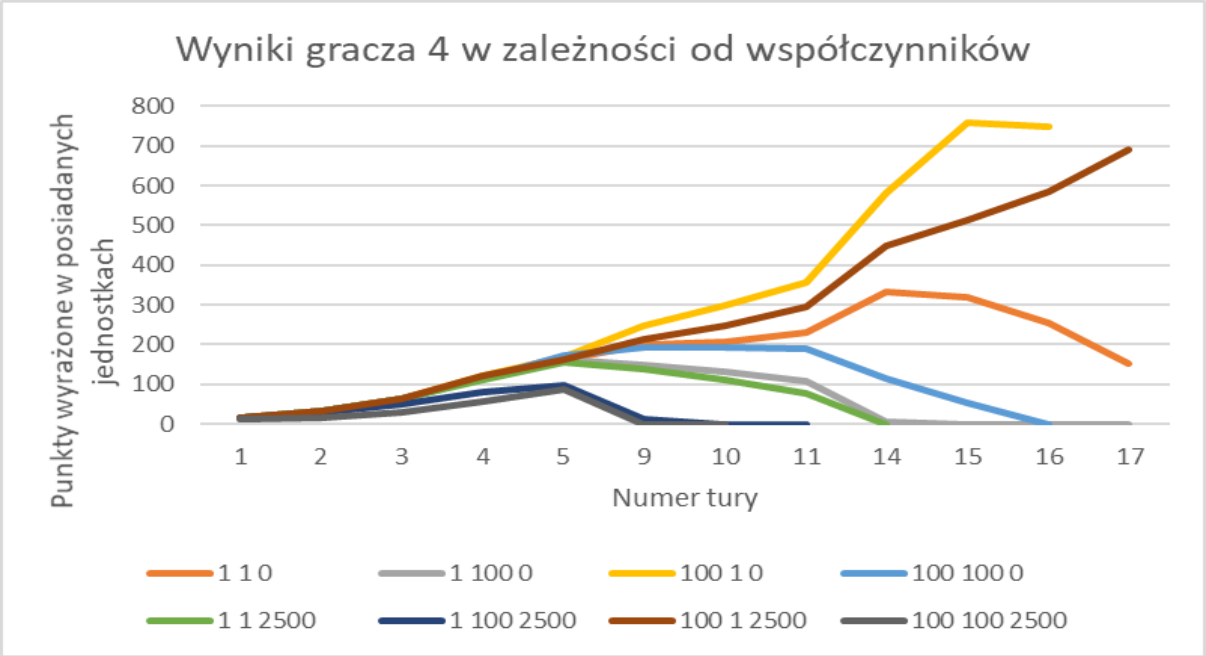


Rysunek 7: Wykres punktów w zależności od tury (opracowanie własne)

Ogólnie według powyższej metodologii wykonano 8 testów (gracz 2 i 3 współczynniki po 50, gracz 1 i 4 współczynniki zmienne), przy dla serii 5-8 gracz 3 i 4 mieli zmieniony współczynnik chęci ataku na 2500. Łącznie gracz 2 wygrał 5 razy, gracz 4 wygrał 2 razy i gracz 3 raz. Ze względu, że badano wyniki gracza 1 i 4, poniżej wykresy ich wyników, gdzie każda seria danych reprezentuje inny zestaw współczynników.

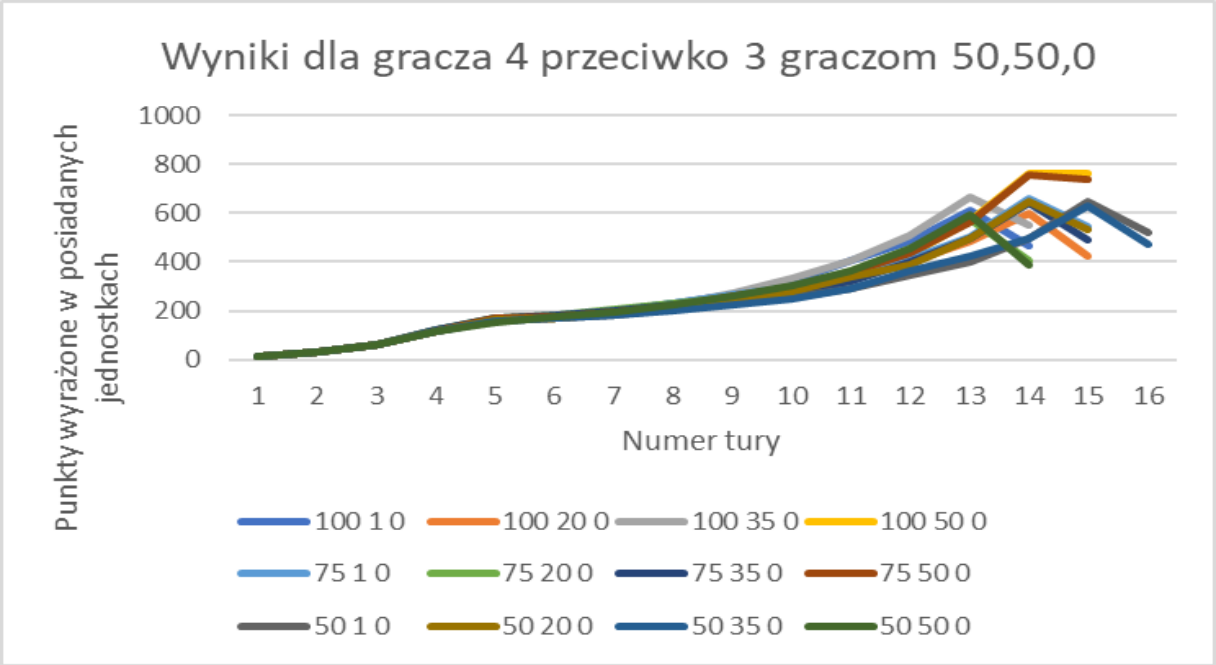


Rysunek 8: Wynik gracza 1 w zależności od współczynników (opracowanie własne)



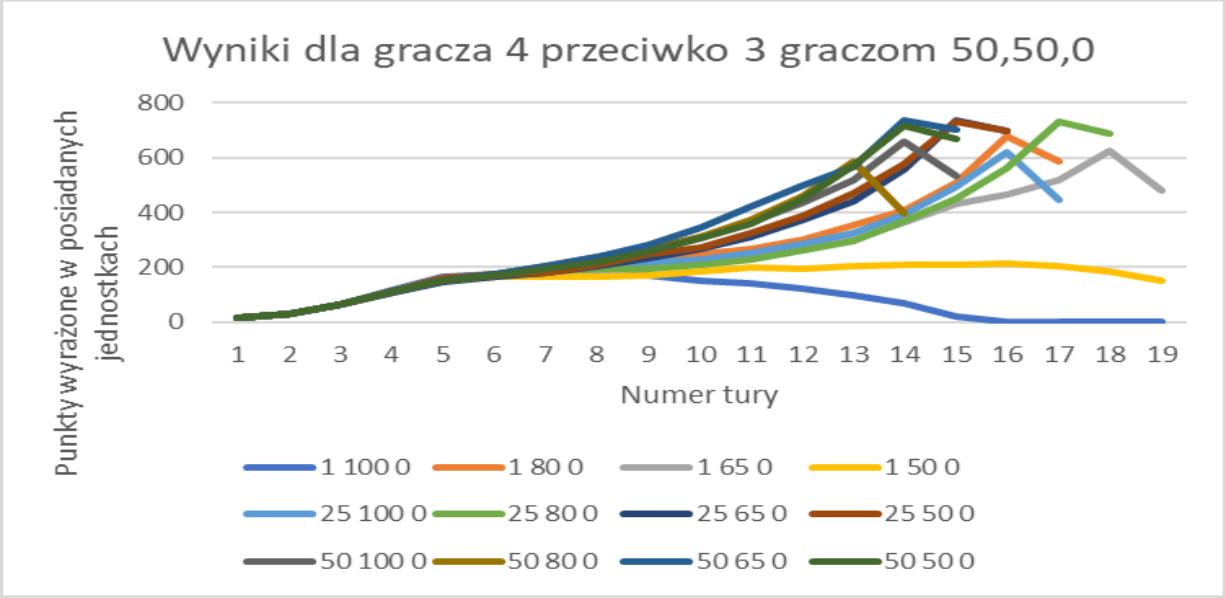
Rysunek 9: Wynik gracza 4 w zależności od współczynników(opracowanie własne)

Z obu powyższych wykresów (Rysunek 8 i Rysunek 9), ale w szczególności z tego z wynikami dla gracza 4, można zobaczyć, że najlepsze wyniki są osiągane dla współczynników 100,1,X, a najgorsze dla 1,100,X. Dodatkowo widać, że dla współczynnika chęci ataku innego niż 0, wzrost jest minimalnie wolniejszy, bo gracz przyjmuje bardziej zachowawczy styl, co pokrywa się z zamierzonym efektem.
Tworzenie kolejnej serii eksperymentów przebiegało w trochę bardziej przemyślany sposób. Rozegrano 12 różnych gier, gdzie gracze 1,2,3 przez cały czas mieli ustawione wartości współczynników na 50,50,0. Dla gracza 4 wartości były zmieniane. Na podstawie wniosków wyciągniętych z pierwszej serii, ustalono że wartości kondensacji będą w zakresie 50-100, a wartości centralizacji w zakresie 1-50. Z racji, że dla podobnych do tych wartości gracz 4 osiągnął najlepsze wyniki w poprzedniej serii, uznano że to dobry sposób na znalezienie najlepszej kombinacji, jako pozytywną weryfikację tezy. Zrezygnowano z modyfikowania współczynnika chęci ataku i ustawiono go stale na 0.



Rysunek 10: Wyniki dla gracza 4 przeciwko 3 graczom 50,50,0, część 1 (opracowanie własne)

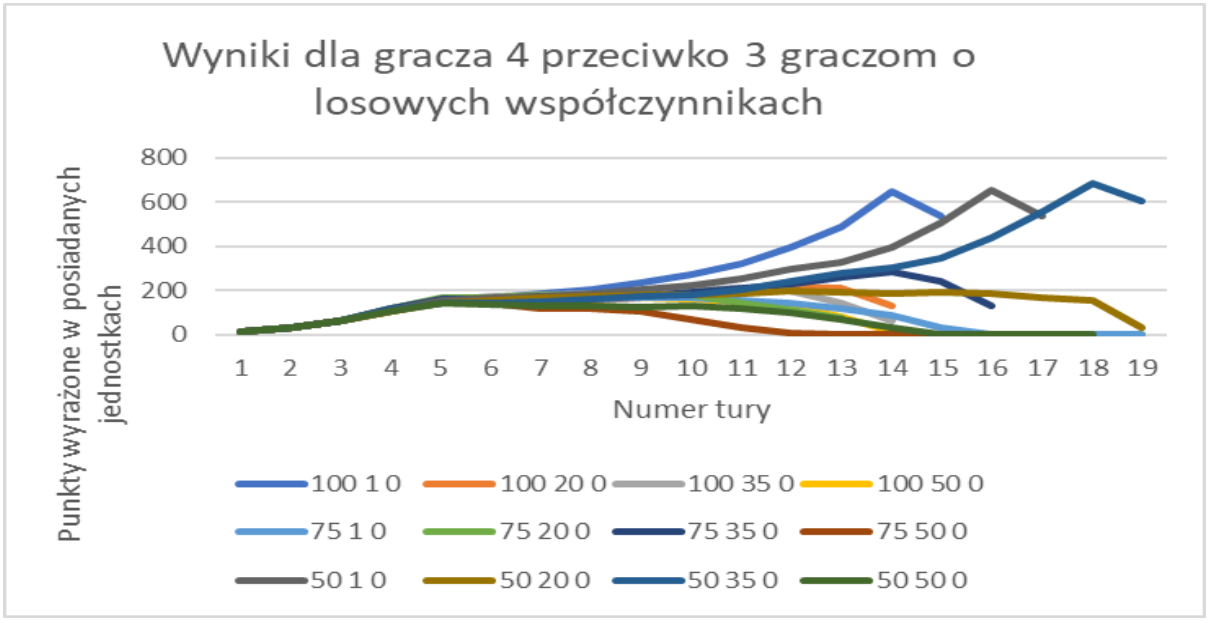
Jak widać gracz 4 wygrał wszystkie rozgrywki (Rysunek 10). Ze względu jednak na sposób sprawdzenia postawionej tezy, ciężko wyciągać z tego większe wnioski. Dla weryfikacji, czy zwycięstwa, nie wynikają z rzeczy niezależnych od współczynników gracza 4, przeprowadzono podobną serię z tą różnicą, że wartości ustawiono na takie przy których oczekiwano by przeciwnych wyników.



Rysunek 11: Wyniki dla gracza 4 przeciwko 3 graczom 50,50,0, część 2 (opracowanie własne)

Widać że tylko dla 2 zestawów współczynników gracz 4 nie odniósł zwycięstwa (Rysunek 11). Poddaje to w wątpliwość tezę, jakoby optimum współczynników dla tego gracza znajdowało się w przedziałach 1-50, 50-100,0. Z racji na brak konkretnej weryfikacji tezy, zdecydowano się na ponowne serie testów dla wcześniejszych wartości współczynników (to z części 1) dla gracza 4, jednak teraz pozostali gracza, mają swoje współczynniki wyznaczane

losowo z przedziału 1-100. Ciężko mówić tutaj już o jakieś konkretnej metodologii, gdyż bardzo dużo zależy tutaj od losowości, jednak jakieś wnioski zawsze da się wyciągnąć.



Rysunek 12: Wyniki dla gracza 4 przeciwko 3 graczom o losowych współczynnikach (opracowanie własne)

Można zauważyć, że gracz 4 wygrał 3 z 12 czyli dokładnie 1/4, co sugerowałoby że jest to w praktyce losowe (Rysunek 12). Ogólnie w ramach tej serii testów zwycięstwa rozkładały się w miarę równo, kolejno: 4,3,2,3. Na podstawie wyników tutaj uzyskanych stworzono tabelę z wartościami współczynników dla każdego zwycięzcy (Tabela 2).

Tabela 2: Współczynniki dla zwycięzców każdego z testów dla 12 testów (opracowanie własne)

gracz	kondensacja	centralizacja
4	100	1
1	99	37
1	55	2
2	73	6
2	87	45
1	70	37
1	74	52
2	53	54
4	50	1
3	36	11
4	50	35
3	88	44
średnia	69,583333	27,08333

Z tabeli można by wywnioskować, że optimum współczynników dla losowego gracza to średnia ze zwycięskich współczynników. Na taki wniosek jest to jednak zbyt mała próbka, aczkolwiek da się tutaj zauważyć wyraźnie trendy w obu przypadkach (kondensacja tylko raz poniżej 50, centralizacja tylko 2 razy powyżej 50).

Z racji braku konkretnego punktu zaczepienia, zdecydowano się wykonać ostatnią serię 20 testów, gdzie wszyscy gracze mają losowe współczynniki i sporządzić tabelę na wzór tej powyżej.

Tabela 3:Współczynniki dla zwycięzców każdego z testów dla 20 testów(opracowanie własne)

gracz	kondensacja	centralizacja
2	77	9
1	83	2
4	72	95
3	88	43
2	92	13
1	83	66
1	74	72
2	81	27
2	77	33
2	21	7
4	78	67
1	81	22
1	66	91
1	79	93
1	19	10
4	35	82
1	77	14
4	68	15
4	82	52
2	94	55
średnia	71,35	43,4
odchylenie standardowe	20,77324	31,41719275

Tak jak wcześniej, próbka jest zbyt mała żeby wyciągać absolutne wnioski, jednak da się zauważyć już pewien trend. Średnia wartość kondensacji w obu seriach testów jest bardzo zbliżona. Z centralizacją jest trochę gorzej, ale w dalszym ciągu wynik nie są od siebie aż tak bardzo oddalone. Na podstawie odchylenia standardowego można wnioskować, że współczynnik kondensacji gra ważniejszą rolę w końcowym wyniku.

5 Podsumowanie

Podsumowując udało się zrealizować wszystkie podstawowe założenia. Stworzono działający systemy gry, który można przystosowywać do różnego rodzaju rozgrywek lub dalszych ulepszeń. Na bazie podstawowego systemu stworzono działającą sztuczną inteligencję, która może stanowić wyzwanie dla gracza ludzkiego. Wykorzystanie elementów biblioteki Pygame umożliwiło w bardzo prosty sposób implementację wszystkich potrzebnych elementów graficzno-funkcyjnych służących do współpracy z użytkownikiem. W procesie testowania sztucznej inteligencji nie osiągnięto do końca oczekiwanego celu, czyli optymalnych wartości współczynników, jednak sensownie zawężono pole poszukiwań.

Projekt może być rozwijany w bardzo wielu kierunkach. Po pierwsze, dodana zostać mogą nowe mechaniki, pokroju specjalnych pól, lub możliwości manipulowania mapą w czasie

rozgrywki. Dodanie wyświetlania krawędzi, na co nie zdecydowano się ze względów estetycznych, umożliwiłoby możliwość usuwania ich podobnie jak usuwane są wierzchołki. Inną ścieżką rozwoju mogłoby być nałożenie na przygotowany zestaw wierzchołków, innych grafik tworzących ciekawą mapę. Jak było wspomniane w pracy, rozwiązanie takie ogranicza trochę system rozgrywki, gdyż jest to rozwiązanie jednorazowe (takiej mapy nie przeniesie się na inny zestaw wierzchołków w łatwy sposób). Ostatnią drogą potencjalnego rozwoju, jest ulepszanie sztucznej inteligencji. Aktualnie działa ona sprawnie, jednak z pewnością można poprawić kilka rzeczy. Na dużych mapach w dalszych fazach rozgrywki komputer potrafi liczyć ruch 2-3 sekundy (dane dla sprzętu autora), co z pewnością da się zoptymalizować. Poza tym poziom gry komputera też zapewne może ulec poprawie, poprzez ulepszanie starych rozwiązań lub dodawanie nowych.

6 Bibliografia

1. Van Rossum G, Drake FL. Python 3 Reference Manual. Scotts Valley, CA: CreateSpace; 2009 https://docs.python.org/3.9/reference/
2. Pete Shinners, PyGame, https://www.pygame.org/docs/ 2020
3. Pyglet team, Pyglet, https://pyglet.readthedocs.io/en/latest/ 2021
4. Tom Rothamel, https://www.renpy.org 2021
5. Igraph development team, https://igraph.org/python/ , 2021
6. NetworkX developers, https://networkx.org , 2021
7. Tiago P. Peixoto, https://graph-tool.skewed.de , 2021
8. PyInstaller Development Team, https://www.pyinstaller.org , 2021
9. Brentvollebregt , https://pypi.org/project/auto-py-to-exe/ , 2021
10. Pablo Pizarro R. Pygame-menu, https://pygame-menu.readthedocs.io/en/4.2.0 2021

7 Spis rysunków

8 Spis tabel

9 Spis listingów