Ewolucja mety gry Hearthstone przy użyciu frameworka SabberStone

(Evolving Hearthstone Meta using the SabberStone Framework)

Kamil Michalak

Praca inżynierska

Promotor: dr Jakub Kowalski

Uniwersytet Wrocławski
Wydział Matematyki i Informatyki
Instytut Informatyki

Streszczenie

Przedmiotem pracy jest analiza metod balansowania gry karcianej Hearthstone, wykorzystujących algorytmy ewolucyjne. Pierwsze rozdziały opisują zasady gry oraz działanie silnika SabberStone, wykorzystanego do symulacji rozgrywek. Dalsza część opisuje istniejące badania nad balansem gry i próbę powtórzenia ich przez autora pracy. Badania te opisują sposoby pomiaru balansu gry, a następnie analizują metody poprawienia go poprzez ewolucyjne modyfikowanie statystyk kart. Powtórzenie opisanych eksperymentów dla innych kart, wykorzystujące autorską implementację, dało podobne, zadowalające wyniki, co wskazuje na skuteczność opisanych metod.

The subject of the thesis is the analysis of Hearthstone card game balancing methods, using evolutionary algorithms. The first chapters describe the game rules and the SabberStone engine used to simulate the games. The next part describes the existing research on the game balance and the author's attempt to replicate it. This research describes how to measure game balance, and then analyzes ways to improve it by evolutionarily modifying card statistics. Repeating the described experiments for other cards, using the proprietary implementation, gave similar, satisfactory results, which indicates the effectiveness of the described methods.

Spis treści

Rozdział 1.

Wstęp

Dzisiejsze gry komputerowe to często bardzo złożone systemy wielu mechanik i obiektów takich jak przedmioty, karty, postacie itp. Złożoność gry umożliwia granie na wiele różnych sposobów i sprawia, że gra jest ciekawsza. Jednak w rywalizacyjnych grach wieloosobowych, gdzie gracze dążą do wygrywania z innymi, prowadzi to zwykle do szukania takich kombinacji obiektów, które dają największe szanse na wygraną. Balansowanie tych obiektów jest ważnym elementem projektowania gier. Jeśli jakaś strategia jest nie do pokonania lub pewne postacie są zbyt słabe, żeby opłacało się nimi grać, to granie w taką grę może być frustrujące i skutkować utratą zainteresowania graczy.

1.1. Cel i zakres pracy

Celem niniejszej pracy jest zbadanie możliwości zbalansowania rozgrywek w kolekcjonerskich grach karcianych (ang. *collectible card games (CCG)*) za pomocą algorytmów ewolucyjnych. Zakres pracy obejmuje:

- Zapoznanie się z mechanikami i zasadami CCG, w szczególności gry *Hearthstone*.
- Zapoznanie się z implementacją framework'a *SabberStone* umożliwiającego symulowanie rozgrywek *Hearthstone*.
- Przegląd badań nad wykorzystaniem algorytmów ewolucyjnych w balansowaniu rozgrywek w CCG, zapoznanie się z wykorzystanymi w nich algorytmami.
- Implementacja przykładowych algorytmów ewolucyjnych do balansowania gier.
- Przeprowadzenie eksperymentów za pomocą zaimplementowanych algorytmów, analiza uzyskanych wyników i wyciągnięcie wniosków.

Rozdział 2.

Kolekcjonerskie gry karciane

Kolekcjonerskie gry karciane to gry, w których gracze zbierają karty do osobistej kolekcji i tworzą z nich talie, którymi rozgrywają mecze przeciwko innym graczom. W grach tego typu, zbudowanie dobrej talii ze współgrającymi ze sobą kartami jest równie ważne co umiejętności gracza. CGG zadebiutowały w formie analogowej wraz z premierą *Magic: The Gathering* [4] w 1993 roku, by z czasem trafić na komputery, konsole i urządzenia mobilne. Obecnie najpopularniejszą cyfrową grą karcianą jest gra *Heartstone* z 23,5 miliona aktywnych graczy w roku 2020. Ponieważ jest ona przedmiotem badań zawartych w tej pracy to w sekcji 2.1. omówione są zasady tej gry.

2.1. Zasady gry Hearthstone

Hearthstone to kolekcjonerska gra karciana wydana przez studio *Blizzard Entertainment* w 2014 roku, na komputery z systemem Windows i Mac OS, oraz urządzenia mobilne z systemem Android i IOS. Gra zawiera ok 2000 grywalnych kart. Talie budowane w grze składają się z bohatera (należącego do jednej z 10 klas, z których każda posiada unikatowe karty i umiejętności), oraz 30 kart (dostępnych dla danej klasy lub neutralnych).

Warunkiem zwycięstwa w meczu jest zredukowanie punktów życia bohatera przeciwnika do 0. Rozgrywka toczy się w systemie turowym. Na początku talie graczy są tasowane i losowane jest, który gracz rozpoczyna rozgrywkę. Następnie gracze dostają karty ze swoich talii - pierwszy gracz dostaje 3 karty, drugi 4 - każdą z nich mogą zachować lub wymienić (fazę wymiany kart określa się jako *mulligan* [3]). W swojej turze gracz zagrywa karty ze swojej ręki, kosztujące kryształy many, których liczba na turę jest ograniczona. Rozgrywka rozpoczyna się z jednym kryształem many u każdego z graczy, po swojej turze gracz otrzymuje dodatkowy kryształ. Maksymalna liczba kryształów to 10.

Istnieją cztery rodzaje kart:

- Stronnik - zagrany pojawia się na stole i może atakować wrogich stronników lub bohatera (zwykle od następnej tury) zadając obrażenie równe posiadanej sile ataku. Zostaje zniszczony gdy otrzyma sumaryczne obrażenia równe lub większe od posiadanej liczby punktów życia. Stronnicy często posiadają dodatkowe efekty np. prowokacja – stronnicy i bohater przeciwnika muszą w pierwszej kolejności atakować stronników z prowokacją, okrzyk bojowy – efekt wywoływany przy zagraniu stronnika.
- Zaklęcie - wywołuje jednorazowy efekt np. zadanie obrażeń, dobranie karty z talii.
- Broń - umożliwia bohaterowi zadawanie obrażeń (równym punktom ataku broni) wrogim stronnikom i bohaterowi, *trwałość* określa ile razy można użyć danej broni przed jej zniszczeniem.
- Bohater - zastępuje bohatera gracza, może zmienić statystyki i umiejętność.

(a) (b) (c) (d)

Rysunek 2.1: Karty: Stronnik (a), Zaklęcie (b), Broń (c), Bohater (d)

2.2. Meta i balans w grach

Meta lub metagra to, w największym skrócie „gra poza grą", jednak ta definicja wymaga rozwinięcia. Przykładowo, jeśli regularnie gramy w Hearthstone'a to po pewnym czasie jesteśmy w stanie dostrzec, że pewne talie pojawiają się częściej, pewne karty są lepsze od innych, przeciwko danej talii lepiej jest grać bardziej lub mniej agresywnie itp. Żeby lepiej poznać te wzorce zaczynamy szukać informacji poza grą np. w poradnikach lub na forach internetowych. Zdobytą w ten sposób wiedzę możemy wykorzystać w grze np. jeśli jakaś talia uchodzi za wyjątkowo mocną to znaczy, że możemy na nią trafiać częściej niż na inne talie, więc warto przygotować się do grania przeciwko tej talii. Metagrę możemy więc zdefiniować jak zbiór czynników spoza zasad gry, które wpływają na doświadczenie płynące z rozgrywki.

Rysunek 2.2: Interfejs Hearthstone'a, źródło: *Hearthstone Wiki* [1]

Jeśli w jakiejś grze rywalizacyjnej jeden sposób grania jest dużo lepszy od innych (w Hearthstone może to być jakaś konkretna talia) to powiemy, że meta jest źle zbalansowana. Takie sytuacje zwykle są niepożądane i wymagają poprawy, dlatego popularne gry multiplayer często aktualizują statystyki kart, przedmiotów, postaci itp.

W grze Hearthstone meta oznacza zwykle zbiór używanych talii, kart i ich statystyk. Ta definicja jest wykorzystywana w dalszej części tej pracy.

Rozdział 3.

Opis silnika SabberStone

SabberStone [5] to silnik do gry Hearthstone napisany na platformie .NET Core w języku C#. Służy on, przede wszystkim, do symulowania rozgrywek. Twórcami silnika jest zespół *HearthSim* [9], który tworzy oprogramowanie związane z Hearthstone'em. Poza Sabberstone'em napisali m.in portal *HSReplay* [6] zbierający powtórki rozgrywek i statystyki.

SabberStone jest wykorzystywany do przeprowadzania zawodów *Hearthstone AI Competition* [7] rozgrywanych corocznie od 2018 roku, o których więcej w sekcji 3.2.3.

3.1. Omówienie implementacji

3.1.1. Struktura klas

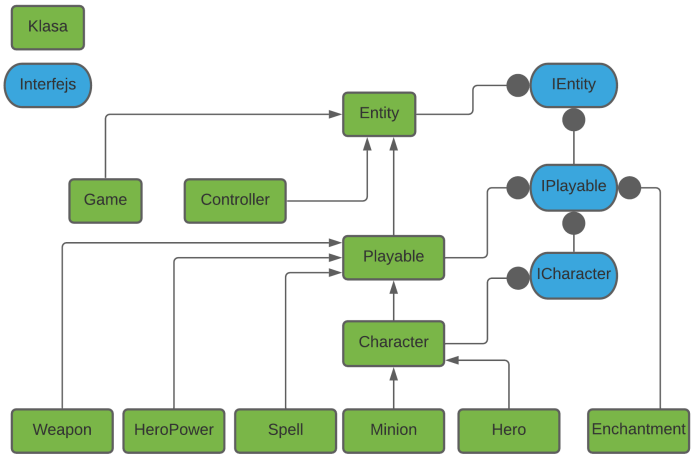
`Entity`

Podstawowa klasa wszystkich obiektów biorących udział w rozgrywce. Wszystkie atrybuty są trzymane w słowniku o kluczach typu `GameTag` i wartościach typu `int`. Klasy dziedziczące po `Entity` są przedstawione na rysunku 3.1. Do utworzenia obiektu klasy `Entity` potrzebny jest obiekt klasy `Card`, o której więcej w sekcji 3.1.2.

`Game`

Reprezentuje pojedynczą rozgrywkę

- `Controller Player1`, `Controller Player2` – gracze
- `Controller CurrentPlayer` – gracz, który aktualnie ma ruch



Rysunek 3.1: Struktura dziedziczenia po klasie `Entity`

- `Controller FirstPlayer` – gracz rozpoczynający rozgrywkę
- `State` – aktualny stan gry (`INVALID/LOADING/RUNNING/COMPLETE`)
- `GameConfig _gameConfig` – konfiguracja gry
- `bool Process(PlayerTask gameTask)` – wykonuje daną akcję gracza (zagranie karty, atak itp) wywołując `gameTask.Process()`.
- `void StartGame(bool stopBeforeShuffling = false)` – metoda wywoływana na początku symulacji. Zmienia `State` na `RUNNING`, ustawia `FirstPlayer` i `CurrentPlayer`. Jeśli `stopBeforeShuffling == false` to tasuje talie obu graczy.
- `void MainReady()` – przygotowuje główną fazę rozgrywki. Trzeba ją wywołać przed wykonaniem pierwszych akcji (nie licząc wymiany kart), inaczej program rzuci wyjątek.
- `string FullPrint()` – zwraca napis opisujący aktualny stan gry (rys. 3.2).

`GameConfig`

Parametr konstruktora klasy `Game`, zawiera konfigurację tworzonej gry: talie i bohaterów obu graczy, informację który gracz zaczyna itp.

`Controller`

Reprezentuje gracza w rozgrywce

```
[ZONE HAND 'FitzVonGerald']|
[HERO]['Garrosh Hellscream[4]']ATK0/AR0/HP30][WP[Rusty Hook[1/3]]][SP0]
[ZONE PLAY 'FitzVonGerald']|[P0][1/1][C1]'N'Zoth's First Mate[32]'|
    [P1][1/1][C1]'Patches the Pirate[20]'|
[ZONE PLAY 'RehHausZuckFuchs']|
[HERO]['Thrall[6]']ATK0/AR0/HP19][WP[NO WEAPON]][SP0]
[ZONE HAND 'RehHausZuckFuchs']|[P0][5/5][C6]'Thing from Below[63]'
    |[P1][5/5][C6]'Thing from Below[62]'|[P2][1/3][C2]'Spirit
Claws[61]'|[P3][1/3][C1]'Tunnel Trogg[67]'|[P4][C0]'The Coin[68]'
    |[P5][1/3][C2]'Spirit Claws[60]'|
```

Rysunek 3.2: Wynik `Game.FullPrint()`

- Listy kart w poszczególnych strefach:
  - `DeckZone` – karty z talii, które nie zostały jeszcze dobrane
  - `HandZone` – karty w ręce
  - `BoardZone` – karty na stole
  - `GraveyardZone` – karty, które zostały zagrane i zniszczone
- `Hero Hero` – bohater gracza
- `int RemainingMana`, `int UsedMana` – pozostałe/zużyte kryształy many w aktualnej turze
- `List<PlayerTask> Options(bool skipPrePhase = true)` – zwraca listę wszystkich akcji możliwych do wykonania przez gracza w aktualnym stanie gry.

Klasy obiektów gry

- `Hero` – bohater
- `Minion` – stronnik
- `Spell` – zaklęcie
- `HeroPower` – umiejętność bohatera
- `Weapon` – broń

`PlayerTask`

Klasa abstrakcyjna reprezentująca akcję wykonaną przez gracza, zawiera abstrakcyjną metodę `void Process()`. Dziedziczą po niej klasy oznaczające konretne typy akcji: zagranie karty (`PlayCardTask`), atak (`MinionAttackTask`, `HeroAttackTask`), koniec ruchu (`EndTurnTask`) itp. Każda z tych klas implementuje metodę `Process`.

(a) (b)

Rysunek 3.3: Karta *Razorfen Hunter* (a) w momencie zagrania przywołuje Dzika (b)

3.1.2. Karty

Klasa `Card` reprezentuje kartę. Karty są wczytywane z pliku `resources/Data/CardDefs.xml`. Dodatkowe efekty kart są dodawane podczas wczytywania przez klasy z katalogu `src/CardSets/` o nazwach kończących się `CardsGen` (np. `CoreCardsGen`). Przykładowo, karta *Razorfen Hunter* (rys. 3.3) kosztuje 3 many (listing 1, wiersz 17), ma 2 punkty ataku (wiersz 16) i 3 punkty życia (wiersz 15), ma też okrzyk bojowy (wiersz 23), który polega na przywołaniu dzika 1/1 (listing 2). Obiekty klasy `Card` są niemodyfikowalne. Podczas rozgrywki zmianie ulegają obiekty dziedziczące po `Entity`, utworzone na podstawie kart.

Statyczna klasa `Cards` umożliwia szybki dostęp do konkretnych kart za pomocą metod:

- `Card FromName(string cardName)`
- `Card FromId(string cardId)`

3.2. Korzystanie z silnika

SabberStone jest bardziej biblioteką niż programem. Nie da się go uruchomić, korzystanie z niego wymaga napisania własnego kodu w C#, który będzie się do tej biblioteki odwoływał. Poboczne projekty znajdujące się w repozytorium SabberStone zawierają przykładowe programy korzystające z silnika.

```
<Entity CardID="CS2_196" ID="257" version="2">
    <Tag enumID="185" name="CARDNAME" type="LocString">
        <enUS>Razorfen Hunter</enUS>
        <plPL>Łowczyni Brzytwoskórych</plPL>
        ...
    </Tag>
    <Tag enumID="184" name="CARDTEXT" type="LocString">
        <enUS><b>Battlecry:</b> Summon a 1/1_Boar.</enUS>
        <plPL><b>Okrzyk bojowy:</b> Przyzwij Dzika 1/1.</plPL>
        ...
    </Tag>
        <Tag enumID="351" name="FLAVORTEXT" type="LocString"> ... </Tag>
        <Tag enumID="365" name="HOW_TO_EARN_GOLDEN" type="LocString"> ... </Tag>
        <Tag enumID="342" name="ARTISTNAME" type="String">Clint Langley</Tag>
        <Tag enumID="45" name="HEALTH" type="Int" value="3"/>
        <Tag enumID="47" name="ATK" type="Int" value="2"/>
        <Tag enumID="48" name="COST" type="Int" value="3"/>
        <Tag enumID="183" name="CARD_SET" type="Int" value="2"/>
        <Tag enumID="199" name="CLASS" type="Int" value="12"/>
        <Tag enumID="201" name="FACTION" type="Int" value="1"/>
        <Tag enumID="202" name="CARDTYPE" type="Int" value="4"/>
        <Tag enumID="203" name="RARITY" type="Int" value="2"/>
        <Tag enumID="218" name="BATTLECRY" type="Int" value="1"/>
        <Tag enumID="251" name="AttackVisualType" type="Int" value="1"/>
        <Tag enumID="321" name="COLLECTIBLE" type="Int" value="1"/>
</Entity>
```

Listing 1: Definicja *Razorfen Hunter* karty w XMLu

```
// ----------------------------------------- MINION - NEUTRAL
// [CS2_196] Razorfen Hunter - COST:3 [ATK:2/HP:3]
// - Fac: horde, Set: core, Rarity: free
// --------------------------------------------------------
// Text: <b>Battlecry:</b> Summon a 1/1_Boar.
// --------------------------------------------------------
// GameTag:
// - BATTLECRY = 1
// --------------------------------------------------------
cards.Add("CS2_196", new CardDef(new Power
{
        PowerTask = new SummonTask("CS2_boar", SummonSide.RIGHT)
}));
```

,

Listing 2: Dodanie okrzyku bojowego (przywołanie dzika) do karty *Razorfen Hunter*, w klasie `CoreCardsGen`

3.2.1. Implementacja rozgrywki pomiędzy AI

SabberStone pozostawia pewną dowolność w sposobie implementacji, ale kod zwykle będzie pasować do szablonu 3. Przy pełnej implementacji należy zwrócić uwagę na kilka rzeczy

- Ostatnia akcja gracza w turze jest zawsze typu `EndTurnTask`
- Statyczna metoda `ChooseTask.Mulligan(Controller controller, List<int> choices)` tworzy akcję (typu `ChooseTask`) wymiany kart o ID z `choices` (właściwość `int Id` z klasy `Entity`). ID kart, które możemy wymienić, znajdują się w `game.Player1.Choice.Choices`. Kartę (typu `IPlayable`) o danym ID otrzymamy dzięki `game.IdEntityDic[id]`.

3.2.2. AI zawarte w projekcie

SabberStone zawiera przykładowe AI w podprojekcie `SabberStoneBasicAI`, które polega na przeszukiwaniu wszerz drzewa gry i ocenie stanu gry za pomocą heurystyki.

Klasa `OptionNode` służy do budowania drzewa.

- `static List<OptionNode> GetSolutions(Game game, int playerId, IScore scoring, int maxDepth, int maxWidth)` – dla danej gry i gracza buduje drzewo o głębokości maksymalnie `maxDepth` i szerokości maksymalnie `maxWidth`, tworząc po kolei poziomy drzewa i zostawiają najwyżej `maxWidth` najlepszych węzłów według metody `scoring.Rate()`. Zwraca listę najlepszych liści.
- `void PlayerTasks(ref List<PlayerTask> list)` – zapisuje do `list` akcje, które należy po kolei wykonać, żeby przejść ze stanu w korzeniu drzewa do stanu w węźle, dla którego metoda jest wywoływana.

Do korzystania z `OptionNode` wymagana jest klasa implementujące interfejs `IScore`(listing 4). Zawiera on deklaracje dwóch metod:

- `int Rate()` – ocena stanu gry, im wyższy wynik tym lepiej.
- `Func<List<IPlayable>, List<int>> MulliganRule()` – zwraca funkcję, która wybiera karty do wymiany w fazie mulligan.

Projekt zawiera kilka klas implementujących `IScore`. Abstrakcyjna klasa `Score` (listing 5) zawiera dodatkowe właściwości ze statystykami stanu gry i trywialne implementacje metod. Klasy dziedziczące po klasie `Score` (np. `AggroScore` – listing 6) korzystają z tych statystyk w nadpisanych wersjach metody `Rate()`. Klasy te implementują standardowe strategie grania, takie jak:

```
var game = new Game(
        new GameConfig()
        {
            Player1Name = ...,   // nazwa gracza: string
            Player1HeroClass = ..., // klasa bohatera: CardClass
            Player1Deck = ..., // talia: List<Card>
            Player2Name = ...,
            Player2HeroClass = ...,
            Player2Deck = ...,
            // Parametry opcjonalne
            StartPlayer = 1/2,  // gracz rozpoczynający, jeśli nie podane to losowo
            FillDecks = true/false, // uzupełnienie talii do 30 kart
            Shuffle = true/false,  // tasowanie talii
            SkipMulligan = true/false,  // pominięcia fazy wymiany kart
            History = true/false,
        });

game.StartGame();

// Jeśli SkipMulligan == false to gracze wymieniają karty
List<int> mulligan1 = ... // Lista ID (IEntity.ID) kart do wymiany
List<int> mulligan2 = ...

game.Process(ChooseTask.Mulligan(game.Player1, mulligan1));
game.Process(ChooseTask.Mulligan(game.Player2, mulligan2));

game.MainReady();

while (game.State != State.COMPLETE)
{
        while (game.State == State.RUNNING && game.CurrentPlayer == game.Player1)
        {
           PlayerTask task = // Wybierz akcję do wykonania
           game.Process(task);
        }

    // analogicznie dla drugiego gracza
}
```

Listing 3: Szablon kodu rozgrywki

- Aggro – strategia dążąca do szybkiego zwycięstwa we wczesnym etapie rozgrywki, przez atakowanie wrogiego bohatera tanimi stronnikami i zaklęciami zadającymi obrażenia, ignorowane wrogich stronników.
- Control – strategia mająca na celu zwycięstwo w późnej fazie gry, używając zaklęć likwidujących stronników przeciwnika, prowokacji i najsilniejszych kart w późniejszych rundach gry.
- Midrange – strategia pośrednia między Aggro i Control, skupia się na zwycięstwie w środkowej fazie gry.

3.2.3. Zasady *Hearthstone AI Competition*

Celem zawodów jest wyłonienie najlepszych agentów grających w Hearthstone, spośród wszystkich przysłanych przez uczestników. Na zawodach odbywają się 2 konkurencje, w jednej gracze używają gotowych talii (6 znanych wcześniej i 3 ukrytych), w drugiej układają własną talię. Obie konkurencje polegają na rozegraniu turnieju w systemie kołowym (każdy z każdym). Każdy mecz składa się z przynajmniej 100 gier a klasyfikacja końcowa zależy od współczynnika zwycięstw każdego z graczy. Agenci w zawodach muszą spełniać kilka reguł:

- Klasa agenta musi dziedziczyć po klasie `AbstractAgent`.
- Czas na wykonanie całego ruchu wynosi 30 sekund, przekroczenie go powoduje przegranie gry.
- Można korzystać z zewnętrznych plików (np. bazy danych), ale ich rozmiar nie może przekroczyć 1GB.
- Agent musi być jednowątkowy.

Interfejs zawodów

W zawodach wykorzystuje się rozszerzenie do SabberStone, które jest dostępne (razem z silnikiem) w repozytorium zawodów [8].

`AbstractAgent` jest bazową klasą dla agentów i wymaga implementacji metod:

- `void InitializeAgent()` – wczytanie plików na początku symulacji
- `void InitializeGame()` – inicjowanie agenta do zagrania nowej gry
- `PlayerTask GetMove(POGame poGame)` – wybór akcji do wykonania w danym stanie gry
- `FinalizeGame()` – zapisanie wyniku i zaktualizowanie agenta po grze

```
public interface IScore
{
        Controller Controller { get; set; }
        Func<List<IPlayable>, List<int>> MulliganRule();
        int Rate();
}
```

Listing 4: Interfejs `IScore`

- `FinalizeAgent()` – zaktualizowanie agenta po symulacji.

Dodatkowo, w konkurencji z własnymi taliami, należy podać talię i klasę bohatera w polach:

- `List<Card> preferedDeck`
- `CardClass preferedHero`

Standardowa klasa `Game` umożliwia dostęp do informacji, które nie są dostępne dla gracza podczas normalnej rozgrywki np. jakie karty są w ręce przeciwnika, dlatego agenci nie mają dostępu do obiektów tej klasy. Zamiast tego korzysta się z klasy `POGame`, która ogranicza ten dostęp. Konstruktor `POGame` przyjmuje obiekt `Game` jako argument, kopiuje go i modyfikuje np. zamienia karty przeciwnika na placeholder'y. Poza tym, klasa zawiera pomocnicze gettery i metodę `Dictionary<PlayerTask, POGame> Simulate(List<PlayerTask> tasksToSimulate)`, która symuluje każdą z akcji z podanej listy w aktualnym stanie gry i zwraca słownik, z obiektami `POGame` uzyskanymi po wykonaniu każdej z akcji.

Klasa `POGameHandler` służy do symulowania gier między dwoma agentami. Metody `PlayGame` i `PlayGames` umożliwiają rozegranie jednej i wielu gier. Statystyki symulacji są przechowywane w obiekcie klasy `GameStats`.

```
public abstract class Score : IScore
{
        public Controller Controller { get; set; }

        public int HeroHp => Controller.Hero.Health;

        public int OpHeroHp => Controller.Opponent.Hero.Health;

        public int HeroAtk => Controller.Hero.TotalAttackDamage;

        public int OpHeroAtk => Controller.Opponent.Hero.TotalAttackDamage;

        public HandZone Hand => Controller.HandZone;

        // ...

        public virtual int Rate()
        {
                return 0;
        }

        public virtual Func<List<IPlayable>, List<int>> MulliganRule()
        {
                return p => new List<int>();
        }
}
```

Listing 5: Klasa `Score`

```
public class AggroScore : Score
{
        public override int Rate()
        {
                if (OpHeroHp < 1)
                        return Int32.MaxValue;

                if (HeroHp < 1)
                        return Int32.MinValue;

                int result = 0;

                if (OpBoardZone.Count == 0 && BoardZone.Count > 0)
                        result += 1000;

                if (OpMinionTotHealthTaunt > 0)
                        result += OpMinionTotHealthTaunt * -1000;

                result += MinionTotAtk;

                result += (HeroHp - OpHeroHp) * 1000;

                return result;
        }

        public override Func<List<IPlayable>, List<int>> MulliganRule()
        {
                return p => p.Where(t => t.Cost > 3).Select(t => t.Id).ToList();
        }
}
```

Listing 6: Klasa AggroScore

Rozdział 4.

Przegląd badań nad ewolucyjnym balansowaniem Hearthstone

Badanie *Evolving the Hearthstone Meta* [10] zostało opublikowane na konferencji *IEEE Conference on Games (CoG) 2019*, która odbyła się w dniach 20-23.08.2019 w Londynie. Celem badania było sprawdzenie jak modyfikacje kart w grze Hearthstone wpływają na strategie i balans gry.

Skupiono się na balansowaniu gry poprzez zmianę czterech atrybutów kart: kosztu w manie, ataku i punktów życia lub trwałości w przypadku kart-broni. Wzmocnienie karty (ang. *buff*) polegało na zmniejszeniu kosztu lub zwiększeniu życia lub ataku. Osłabienie (ang. *nerf*) – na zwiększeniu kosztu lub zmniejszeniu życia/ataku. Zdefiniowano wielkość zmian jako sumę wartości bezwzględnych zmian atrybutów pomnożonych przez ich wagę.

$$\sum |C_i| \cdot w_i$$

Gdzie C_i to wartość zmiany atrybutu, a waga w_i jest równa 2 dla zmiany w manie i 1 w pozostałych przypadkach. Zrobiono tak ponieważ zmiana kosztu o 1 jest w praktyce bardziej znacząca niż zmiana ataku/życia o 1.

Wykorzystano talie z badania wyewoluowane w badaniu [12] – po 2 wersje talii dla klas Łowca (ang. *Hunter*), Paladyn (ang. *Paladin*) i Czarnoksiężnik (ang. *Warlock*), korzystających ze strategii Aggro i Control, razem 12 talii. Następnie rozegrano 10000 rozgrywek między każdymi dwiema taliami, żeby obliczyć skuteczność danych talii. Wybrano 3 talie: Łowca (Aggro), Paladyn (Control) i Czarnoksiężnik (Control) do przeprowadzenia dalszych eksperymentów.

4.1. Eksperyment 1: ewolucja mety

Pierwszy eksperyment polegał na próbie zbalansowania rozgrywki poprzez ewolucje statystyk kart. Trzy talie zawierały w sumie 64 unikatowe karty: 56 stronników, 4 zaklęcia i 2 bronie. W zaklęciach modyfikowano koszt a w stronnikach i broniach: koszt, atak i punkty życia.

Populacja składała się z chromosomów reprezentujących zmiany w kartach – wektorów liczb całkowitych o długości $58 \cdot 3 + 6 \cdot 1 = 180$. Każdy gen chromosomu oznaczał zmianę jednej właściwości karty o wartość z przedziału $[-3, 3]$. W tym eksperymencie, niezależnie od chromosomów, każda właściwość zawierała się w przedziale $[0, 10]$

Parametry algorytmu to: populacja 100 osobników, współczynnik krzyżowania (ang. *crossover rate*) = 35%, współczynnik mutacji (ang. *mutation rate*) = 20%. Rodzice byli wybierani za pomocą turnieju o rozmiarze 3, mutacja polegała na zmianie każdego genu na losową wartość z prawdopodobieństwem 5%. Ocena osobników polegała na rozegraniu po 100 rozgrywek między każdymi dwiema taliami. Funkcja oceny miała postać

$$F = \sqrt{\frac{4}{3}\sum(w_{ij} - 0,5)^2}$$

Gdzie w_{ij} to współczynnik zwycięstw talii i z talią j. Dla kart idealnie zbalansowanych $F = 0$. Balans talii bez modyfikacji wyniósł 0,43. W 12. pokoleniu najlepszy osobnik osiągnął wynik 0,0062, ale jego wielkość zmian wynosiła aż 403 przy średnim wyniku 402 dla tego pokolenia (gdzie maksymalny możliwy to 540).

4.2. Eksperyment 2: ewolucja wielokryterialna

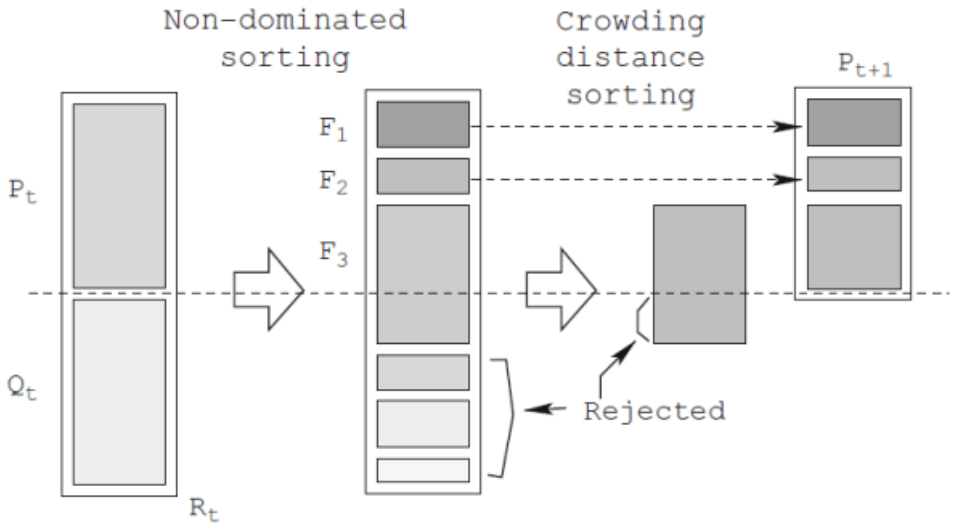
Celem drugiego eksperymentu było zmaksymalizowanie balansu przy jednoczesnym zminimalizowaniu zmian w kartach. Wykorzystano do tego algorytm NSGA2 [11]. Jest to algorytm ewolucyjny szukający najlepszych osobników na podstawie więcej niż jednego kryterium, w tym przypadku dwóch: balansu i wielkości zmian. Wykorzystana jest wielokryterialna funkcja oceny (ang. *multi-objective fitness, MOF*) zdefiniowana następująco:

$$MOF_i = (B_i, M_i)$$

Gdzie B to funkcja balansu gry a M_i to wielkość zmian w i-tym chromosomie.
Osobnik i **dominuje** osobnika j jeśli

- $B_i \leq B_j$ i $M_i \leq M_j$ i $(B_i < B_j$ lub $M_i < M_j)$

Niezdominowany zbiór/front to podzbiór populacji, w którym żaden osobnik nie dominuje innego.



Rysunek 4.1: Iteracja w NSGA2

Obliczanie następnej populacji

1. z aktualnej populacji P_t, zawierającej n osobników utwórz populację potomną Q_t o tym samym rozmiarze. Niech $R_t = P_t \cup Q_t$.

2. Niedominowane sortowanie – podziel R_t na niedominowane fronty $F_1, F_2, \ldots, F_k$. W F_1 znajdą się wszystkie osobniki niezdominowane przez żadne inne (F_1 stanowi front Pareta populacji R_t), w F_2 te zdominowane tylko przez niektóre z F_1, w F_3 – tylko przez F_1 i F_2 itd. Każdemu osobnikowi nadaj ranking równy numerowi frontu.

3. Utwórz populację P_{t+1} biorąc n najlepszych osobników z R_t według rankingu. Jeśli front F_i mieści się tylko częściowo w populacji to posortuj jego elementy malejąco, według odległości do „sąsiadów" z frontu (ang. *crowding-distance*), i wybierz najlepsze do nowej populacji.

Początkowa populacja składała się ze 100 chromosomów: 98 losowych, najlepszego osobnika z eksperymentu 1 i takiego, który nie wprowadza żadnych zmian w kartach (o balansie 0,43). Pozostałe parametry ewolucji były takie jak w eksperymencie 1. Wykonano dwie osobne ewolucje dla 32 i 47 pokoleń. Najlepszy osobnik osiągnął balans 0,008 przy zmianach 154. W porównaniu do eksperymentu 1, osiągnięto podobny balans rozgrywki (0,008 vs 0,006) mocno ograniczając zmiany w kartach (154 vs 402), więc udało się osiągnąć cel eksperymentu.

4.3. Eksperyment 3: heurystyka wyboru kart do balansowania

Celem eksperymentu 3 było zbadanie, które kart należy wzmocnić lub osłabić, żeby osiągnąć jak najlepszy balans przy jak najmniejszych zmianach. W tym celu wykorzystano trzy współczynniki mierzone dla każdej karty z talii:

- współczynnik zwycięstw jeśli zagrana (ang. *Win Rate when Played*, WRP), liczony jako liczba zwycięstw, gdzie przynajmniej jedna kopia karty była zagrana, podzielona przez liczbę gier gdzie karta była zagrana
- współczynnik zwycięstw jeśli dobrana (ang. *Win Rate When Drawn*, WRD) – liczba zwycięstw gdzie przynajmniej jedna kopia karty była dobrana z talii, podzielona przez liczbę gier gdzie karta była dobrana
- współczynnik zwycięstw po osłabieniu (ang. *Win Rate After Nerf*, WRN) – współczynnik zwycięstw talii jeśli dana karta ma koszt zwiększony o 1.

z 12 talii z [12] wybrano talię o największym współczynniku zwycięstw (Paladyn (Control)). Następnie obliczono WRN dla każdej karty, zwiększając jej koszt i rozgrywając 10000 gier przeciwko pozostałym 11 taliom. Postawiono hipotezę, że zwiększenie kosztu karty z wyższym WRP/WRD będzie miało większy (negatywny) wpływ na współczynnik zwycięstw talii, spodziewano się też, że WRD będzie miało większy wpływ niż WRP.

Zaobserwowano niewielką negatywną korelację między WRP i WRN oraz nieco większą między WRD i WRN, osłabienie kart z wyższym WRP lub WRD spowodowało większy spadek współczynnika zwycięstw zbyt silnej talii. Według autorów, uzyskane wyniki zdawały się wskazywać na słuszność początkowych przypuszczeń.

4.4. Wnioski

Opisana publikacja zaprezentowała metody do pomiaru balansu w grach karcianych oraz metody balansowania tych gier. Eksperyment 1 pokazał, że algorytmy ewolucyjne umożliwiają znalezienie modyfikacji kart dających prawie idealnie zbalansowaną rozgrywkę. W eksperymencie 2, dzięki algorytmowi NSGA2 uzyskano bardzo podobny balans przy o wiele mniejszych zmianach w kartach. W końcu, w eksperymencie 3 zaproponowano sposoby mierzenia siły kart i wskazywania, które karty należy osłabić, żeby talia była istotnie słabsza. Autorzy zadeklarowali prowadzenie w przyszłości dalszych badań na ten temat.

Rozdział 5.

Własna implementacja badań

Poniższy rozdział opisuje autorskie przeprowadzenie eksperymentów takich jak w publikacji [10] opisanej w rozdziale 4., wykorzystując inne talie i strategie, i porównanie wyników. W autorskiej implementacji eksperymentów wykorzystano talie podstawowe [2] (domyślne dla nowych graczy) dla klas Druida, Łowcy i Maga, każdą w 3 wersjach strategii: Aggro, Control i Midrange.

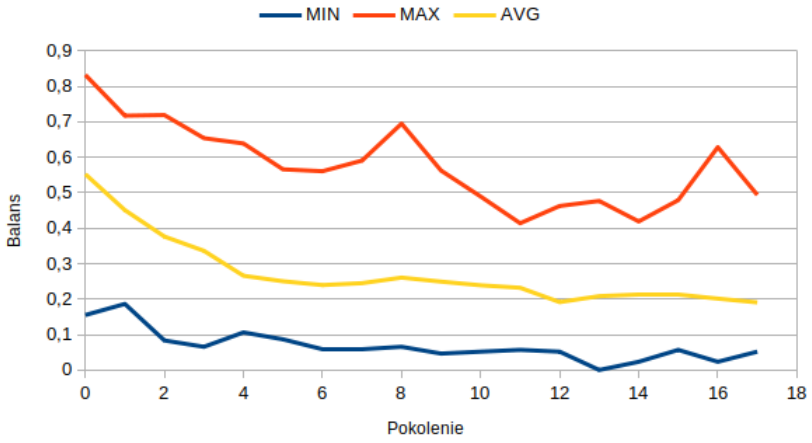
W celu sprawdzenia siły talii rozegrano po 50 gier każdy z każdym. Wyniki przedstawiono w tabeli 5.1. Do dalszych eksperymentów wybrano talie: Łowca (Aggro), Druid (Control), Mag (Midrange).

5.1. Eksperyment 1: ewolucja mety

Celem pierwszego eksperymentu było osiągnięcie jak najlepszego balansu rozgrywki dla trzech podstawowych talii. Każda podstawowa talia (dla wszystkich 10 klas) zawiera 5 kart klasy, dostępnych dla nowych graczy oraz 10 kart spośród podstawowych kart neutralnych. Każda karta z talii występuje w 2 kopiach. W taliach Łowcy, Druida i Maga jest w sumie 35 unikatowych kart: 22 stronników i 13 zaklęć.

	Druid (Aggro)	Druid (Control)	Druid (Midrange)	Mag (Aggro)	Mag (Control)	Mag (Midrange)	Łowca (Aggro)	Łowca (Control)	Łowca (Midrange)	wsp. zwycięstw vs meta
Druid (Aggro)	0,50	0,18	0,18	0,26	0,06	0,02	0,32	0,06	0,02	0,18
Druid (Control)	0,82	0,50	0,54	0,80	0,44	0,34	0,72	0,42	0,34	0,55
Druid (Midrange)	0,82	0,46	0,50	0,98	0,60	0,58	0,86	0,42	0,36	0,62
Mag (Aggro)	0,74	0,20	0,02	0,50	0,00	0,06	0,58	0,04	0,00	0,24
Mag (Control)	0,94	0,56	0,40	1,00	0,50	0,18	1,00	0,34	0,22	0,57
Mag (Midrange)	0,98	0,66	0,42	0,94	0,82	0,50	1,00	0,50	0,46	0,70
Łowca (Aggro)	0,68	0,28	0,14	0,42	0,00	0,00	0,50	0,02	0,02	0,22
Łowca (Control)	0,94	0,58	0,58	0,96	0,66	0,50	0,98	0,50	0,3	0,67
Łowca (Midrange)	0,98	0,66	0,64	1,00	0,78	0,54	0,98	0,7	0,50	0,75

Tablica 5.1: Wyniki meczów między taliami



Rysunek 5.1: Maksymalna, minimalna i średnia wartość funkcji oceny dla 17 populacji

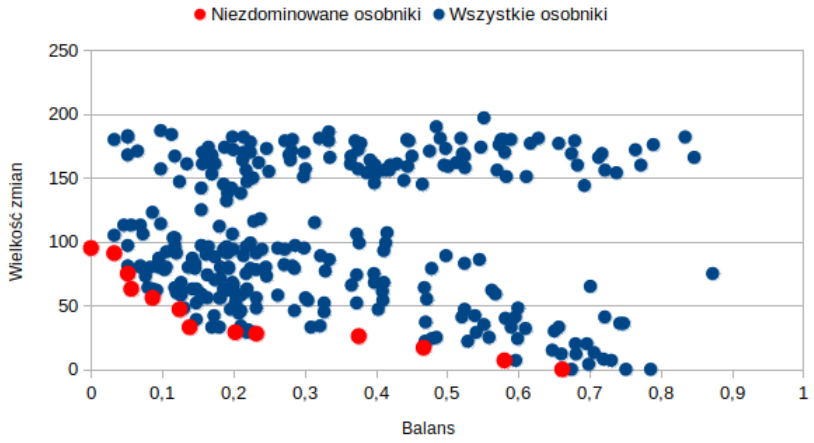
Chromosomy mają więc długość $22 \cdot 3 + 13 \cdot 1 = 79$. Ze względu na czas trwania symulacji rozgrywek ograniczono populację do 50 osobników, oraz liczbę symulacji podczas oceny do $3 \cdot 50 = 150$ Pozostałe parametry ewolucji pozostały identyczne jak w [10].

Wyniki eksperymentu są przedstawione na wykresie 5.1. W 13 pokoleniu jeden chromosom osiągnął idealny balans. Jednak, ze względu na małą liczbę symulacji, funkcja oceny nie była zbyt dokładna. W szczególności niektóre osobniki były ewaluowane kilka razy w różnych pokoleniach i uzyskiwały dość istotnie różne wyniki. Dlatego następnie wzięto 10 najlepszych osobników z całego eksperymentu i policzono ich balans symulując 500 gier dla każdej pary talii. Najlepszy – inny od tego który osiągnął zero podczas ewolucji – osiągnął wynik 0,057 przy zmianach 182 (gdzie maksimum to 342) i pojawił się po raz pierwszy w 9. pokoleniu. Balans niezmodyfikowanych kart wyniósł 0,686.

Osiągnięty wynik jest istotnie gorszy od tego z [10]. Może to być spowodowane kilkoma czynnikami: użyciem innych talii, ograniczeniem wielkości populacji i liczby symulacji podczas oceny, czy niezamierzonych różnic w implementacji i działaniu algorytmu. Mimo wszystko wykres 5.1 pokazuje widoczną poprawę wyników wraz z czasem działania algorytmu. Należałoby zastanowić się nad poprawieniem implementacji algorytmu i uruchomić tę poprawioną wersję na mocniejszym komputerze.

5.2. Eksperyment 2: ewolucja wielokryterialna

W eksperymencie 1 udało się znaleźć takie modyfikacje kart, które istotnie przybliżają talie do 50% zwycięstw, chociaż, na podstawie [10], spodziewano się nieco lepszych rezultatów. Kolejnym celem było znalezienie rozwiązań, które balansują rozgrywkę w podobnym (lub większym), stopniu a jednocześnie wprowadzają mniej



Rysunek 5.2: Wykres punktowy osobników z ewolucji wielokryterialnej

zmian w kartach. Zaimplementowano w tym celu algorytm NSGA2, opisany w rozdziale 4. i [11].

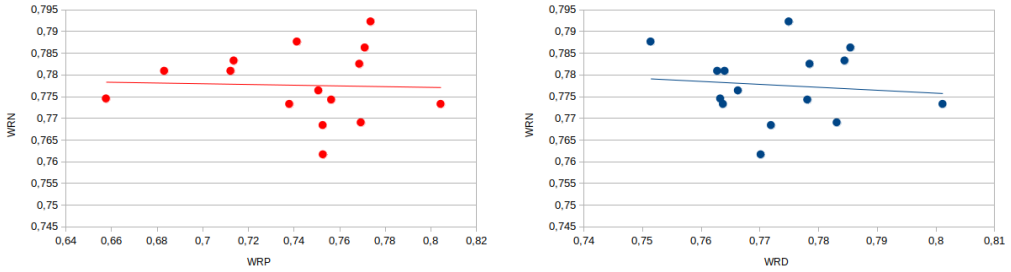
Początkową populację utworzono z 48 losowych rozwiązań, najlepszego rozwiązania z eksperymentu 1 i takiego, które w ogóle nie modyfikuje kart. Zmieniono nieco sposób tworzenia nowej populacji względem tego opisanego [10] – populację Q_t (rys. 4.1) tworzono w całości poprzez krzyżowanie (zamiast 35%), najlepsze osobniki z P_t i tak trafiały do P_{t+1} dzięki sortowaniu. Mutacja Q_t wyglądała już identycznie jak we wspomnianym badaniu.

Algorytm uruchomiono dla 25 pokoleń, wszystkie uzyskane osobniki przedstawiono na wykresie 5.2. Następnie, ze wszystkich pokoleń, wybrano 20 osobników ze wskaźnikiem zmian poniżej 100 i jak najlepszym balansie do dokładniejszej oceny ($3 \cdot 500$ gier). Najlepsze rozwiązanie osiągnęło balans 0,074 przy zmianach 95, drugie najlepsze osiągnęło niewiele gorszy balans równy 0,075 przy zmianach jedynie 63. Udało się znaleźć, rozwiązania niewiele gorsze od tych z eksperymentu 1, a wprowadzające o wiele mniej zmian w kartach.

5.3. Eksperyment 3: heurystyka wyboru kart do balansowania

Celem eksperymentu 3 było sprawdzenie czy zaproponowane w [10] wskaźniki WRP i WRN faktycznie pomagają przy wyborze kart do modyfikacji. Na podstawie tabeli 5.1 wybrano najsilniejszego „gracza” – Łowca (Midrange) – i rozegrano po 1000 gier z każdym z pozostałych 8 graczy, obliczając WRP i WRD każdej karty z talii. Następnie obliczono WRN każdej karty zwiększając jej koszt o 1 i symulując 8000 gier.

W tabeli 5.2 przedstawiono obliczone WRP, WRD i WRN dla wszystkich kart



(a) WRP vs WRN (b) WRD vs WRN

Rysunek 5.3: Wykresy zależności między WRP/WRD a WRN, z prostymi regresji liniowej

z talii. Kartą o najwyższej wartości, zarówno WRP jak i WRD, okazał się Treser Ogarów (ang. *Hundmaster*, rys. 5.2b) – stronnik kosztujący 4 punkty many ze statystykami 4/3 i okrzykiem bojowym dającym +2/+2 wybranemu stronnikowi, oznaczonemu jako bestia. Współczynnik zwycięstw talii dla tych 8000 gier wyniósł 0,793.

Wykresy na rys. 5.3 przedstawiają zależności między WRP i WRD a WRN razem z prostymi regresji. Podobnie jak w [10] zaobserwowano minimalną negatywną korelację między WRP i WRN, oraz minimalnie większą między WRD i WRN. Wydaje się, że w przeprowadzonym eksperymencie WRP miało nieco większe znaczenie, a WRD trochę mniejsze w porównaniu do wyników z [10], co najprawdopodobniej wynika z użycia zupełnie innych talii.

Nazwa	WRP	WRD	WRN
Arcane Shot	0,741	*0,751*	0,788
Bloodfen Raptor	0,769	0,783	0,769
Core Hound	0,714	0,784	0,783
Houndmaster	**0,804**	**0,801**	0,773
Ironforge Rifleman	0,738	0,764	0,773
Multi-Shot	*0,658*	0,763	0,775
Oasis Snapjaw	0,756	0,778	0,774
Raid Leader	0,774	0,775	**0,792**
Razorfen Hunter	0,753	0,770	*0,762*
River Crocolisk	0,771	0,785	0,786
Silverback Patriarch	0,712	0,763	0,781
Stonetusk Boar	0,751	0,766	0,776
Stormpike Commando	0,753	0,772	0,768
Timber Wolf	0,769	0,778	0,783
Tracking	0,683	0,764	0,781

(a)

(b) Treser Ogarów

Tablica 5.2: Wartości WRP, WRD i WRN dla kart z talii. Pogrubieniem oznaczono największe wartości w kolumnach a kursywą najmniejsze.

5.4. Wnioski

Wyniki eksperymentów opisanych w tym rozdziale zdają się potwierdzać wnioski autorów badania [10] opisanego w rozdziale 4. Różnice w uzyskanych wynikach są widoczne, ale należało się ich spodziewać ze względu na różnice w implementacji i użytych taliach. W wynikach jednego i drugiego badania można znaleźć pewne wspólne cechy.

- Stosunek balansu najlepszego osobnika z eksperymentu 1 do najlepszego osobnika z eksperymentu 2 jest prawie identyczny.
- W eksperymencie 3 WRD okazało się bardziej znaczące niż WRP

Eksperymenty należałoby powtórzyć, ale wydaje się, że opisane metody działają.

Rozdział 6.

Podsumowanie

Celem niniejszej pracy było zbadanie metod balansowania kolekcjonerskiej gry karcianej Hearthstone za pomocą algorytmów ewolucyjnych. Wszystkie cele, opisane we wstępie pracy (rozdział 1.), zostały osiągnięte

W rozdziale 2. przedstawiono zasady i mechaniki gry Hearthstone. Opisano przebieg rozgrywki, omówiono rodzaje kart, ich statystyki i specjalne umiejętności. W sekcji 2.2. wyjaśniono czym jest metagra.

W rozdziale 3. omówiono silnik SabberStone do symulowania gier w Hearthstone. W sekcji 3.1. opisano główne elementy silnika:

- strukturę klas reprezentujących elementy gry, najważniejsze metody i właściwości tych klas.
- implementację kart, ich definicje w formacie xml, wczytywanie ich z pliku xml do obiektu klasy `Card`, tworzenie obiektów w rozgrywce z obiektów `Card`

W sekcji 3.2. zaprezentowano jak wykorzystywać silnik do przeprowadzania symulacji i opisano AI dostępne w pobocznym projekcie. Omówiono też zawody *Hearthstone AI Competition*, wykorzystywane w nich rozszerzenie do SabberStone'a i wymagania stawiane grającym agentom.

W rozdziale 4. omówiono badania [10] nad wykorzystaniem algorytmów ewolucyjnych do balansowania rozgrywki Hearthstone. Opisano przedstawiony sposób pomiaru balansu gry, reprezentację zmian w kartach oraz miarę wielkości tych zmian. W pierwszym z opisanych eksperymentów znaleziono ewolucyjnie rozwiązanie dające dobrze zbalansowaną grę, ale rozwiązanie to wymagało wprowadzenia wielu zmian w kartach. W eksperymencie 2 wykorzystano algorytm ewolucji wielokryterialnej NSGA2 i znaleziono rozwiązanie, które dawało podobnie dobry balans, ale wprowadzało o wiele mniej zmian w kartach. W końcu, w eksperymencie 3 zaproponowano metody wyboru kart do modyfikacji i zaobserwowano, że w pewnym stopniu działają.

W rozdziale 5. omówiono autorską implementację eksperymentów z rozdziału 4. Te same metody balansu rozgrywki zastosowano dla innych talii kart. Ze względu na czas trwania obliczeń, ograniczono liczbę symulacji i rozmiar populacji w eksperymentach. Uzyskane wyniki były podobne do tych uzyskanych w [10], co sugeruje, że opisane metody działają

Opisane badania sugerują, że algorytmy ewolucyjne mają potencjał do wykorzystania w rozwoju gier wideo. W tej pracy skupiono się na grach karcianych, ale wydaje się, że opisane sposoby balansowania rozgrywki można wykorzystać również w innych grach. W opisanych badaniach oceniano balans gry przeprowadzając symulacje za pomocą dość prostego AI, ale studia developerskie mogą wykorzystywać do tego graczy testujących grę na oddzielnych serwerach testowych, co obecnie dzieje się np. w grze *League of Legends* [15]. Poruszony w tej pracy temat jest warty dalszych badań, zarówno w kontekście prac akademickich, jak i rzeczywistego wdrażania go w proces produkcji gier.

Bibliografia

[1] Hearthstone Wiki - kompendium wiedzy o grze Hearthstone
https://hearthstone.gamepedia.com/Hearthstone_Wiki

[2] https://hearthstone.gamepedia.com/Basic_deck

[3] https://hearthstone.gamepedia.com/Mulligan

[4] Oficjalna strona *Magic: The Gathering*
https://magic.wizards.com/en

[5] Repozytorium SabberStone
https://github.com/HearthSim/SabberStone

[6] https://hsreplay.net/

[7] https://hearthstoneai.github.io

[8] https://github.com/ADockhorn/HearthstoneAICompetition

[9] HearthSim
https://hearthsim.info/

[10] F. de Mesentier Silva, R. Canaan, S. Lee, M. C. Fontaine, J. Togelius, A. K. Hoover, *Evolving the Hearthstone Meta* IEEE Conference on Games 2019
https://arxiv.org/abs/1907.01623

[11] K. Deb, S. Agrawal, A. Pratap, and T. Meyarivan, *A fast elitist nondominated sorting genetic algorithm for multi-objective optimization: Nsga-ii*, International conference on parallel problem solving from nature. Springer, 2000, pp. 849–858.
https://www.iitk.ac.in/kangal/Deb_NSGA-II.pdf

[12] Matthew C. Fontaine, Scott Lee, L.B. Soros, Fernando De Mesentier Silva, Julian Togelius, Amy K. Hoover, *Mapping Hearthstone Deck Spaces through MAP-Elites with Sliding Boundaries*
https://arxiv.org/abs/1904.10656

[13] Institute of Electrical and Electronics Engineers
https://www.ieee.org/

[14] IEEE Conference on Games 2019
https://ieee-cog.org/2019/

[15] Oficjalna strona *League of Legends*
https://na.leagueoflegends.com/pl-pl/