

UNIwersYTET RZESZOWSKI
Kolegium Nauk Przyrodniczych



Albert Mazur

117816

Informatyka

**Projekt i implementacja gry karcianej opartej o grę Uno z wykorzystaniem algorytmów
heurystycznych**

Praca inżynierska

Praca wykonana pod kierunkiem

dra inż. Piotra Laska

Rzeszów, 2024

Spis treści

1. Wstęp.....	4
1.1. Wprowadzenie.....	4
1.2. Wybór tematu.....	4
1.3. Cel i zakres pracy	4
2. Sztuczna inteligencja w grach komputerowych	6
2.1. Zastosowanie algorytmów AI w grach.....	6
2.2. Przykładowe algorytmy	7
2.2.1. Algorytm A*	7
2.2.2. Maszyny Stanów Skończonych	8
2.2.3. Logika rozmyta.....	10
3. Projekt.....	15
3.1. Architektura	15
3.2. Wykorzystane technologie	15
3.3. Baza danych	16
3.3.1. Schemat	16
3.3.2. Opis tabel.....	16
4. Implementacja	18
4.1. Opis interfejsu użytkownika	18
4.2. Algorytm o ustalonych zasadach	24
4.2.1. Opis algorytmu.....	24
4.2.2. Schemat blokowy i opis implementacji	24
4.3. MCTS (Monte Carlo Tree Search)	27
4.3.1. Opis algorytmu.....	27
4.3.2. Opis implementacji	27
4.4. Porównanie algorytmów	34
5. Zakończenie	36
5.1. Podsumowanie i wnioski	36
5.2. Dalsze możliwości rozwoju aplikacji	36
Bibliografia.....	38
Spis rysunków i kodów źródłowych.....	39

Streszczenie

Praca inżynierska dotyczy implementacji i stworzenia gry karcianej opartej za zasadach podobnych jak w grze Uno. Wykorzystując dwa algorytmy heurystyczne pierwszy oparty o ustalone zasady opisane w pracy i drugi Monte Carlo Tree Search (MCTS). Praca skupia się na zastosowaniu sztucznej inteligencji w grach, z naciskiem na algorytmy AI, które są w stanie wygrać z człowiekiem. Opisuje także architekturę projektu, użyte technologie, bazę danych oraz implementację interfejsu użytkownika. Porównując efektywność MCTS z algorytmem wybierający kartę z ustalonych zasad opisanych w pracy dla gry podobnej do Uno. Na koniec zostały przedstawione wnioski i możliwości dalszego rozwoju aplikacji.

Angielski tytuł pracy

Design and Implementation of an Uno-Based Card Game with Heuristic Algorithms

1. Wstęp

1.1. Wprowadzenie

W dobie rosnącego trendu na stosowanie sztucznej inteligencji we wszelakich dziedzinach życia - w tym mas mediów, użytkownicy mają sposobność doświadczania utworów kulturalnych wspomaganych przy użyciu AI [1]. Wszechstronność i dowolność przy jego zastosowaniu dała ludziom nowe możliwości, w tym realizację klasycznych metod spędzania wolnego czasu w gronie znajomych [2]. W pracy inżynierskiej przedstawiono, opis projektu gry karcianej wspomaganej o algorytmy sztucznej inteligencji.

1.2. Wybór tematu

W pracy inżynierskiej wybrano do implementacji grę Uno, ponieważ charakteryzuje się ona podstawową zasadą: gracze muszą dopasowywać karty pod względem koloru lub numeru do karty poprzednio zagranej przez przeciwnika. W czasie rozgrywki gra przewiduje różnorodne możliwości w tym m. in.: dodawania kart do ręki jednego z graczy w ramach kary, zmniejszenie jej ilości i przenoszenie na kolejnego gracza. Te zróżnicowane opcje dają graczom możliwość samodzielnego podejmowania strategii, które są jednak częściowo zdeterminowane przez elementy losowości.

Do implementacji bota w grze wybrano podejście heurystyczne, ponieważ analizowanie całej przestrzeni rozwiązań mogło być zbyt czasochłonne. Heurystyka to sposób rozwiązywania problemów, który wykorzystuje praktyczne podejście do znalezienia wystarczająco dobrego rozwiązania w rozsądnym czasie, szczególnie w sytuacjach, gdy pełne przeszukiwanie przestrzeni rozwiązań jest niemożliwe, zbyt czasochłonne lub niepraktyczne. Algorytmy heurystyczne nie gwarantują znalezienia optymalnego rozwiązania, ale często prowadzą do dobrego lub przynajmniej akceptowalnego rozwiązania [3].

Użyto dwa algorytmy heurystyczne: Monte Carlo Tree Search i wybór karty o ustalonych zasadach opisanych poniżej. MCTS, znany jest z efektywności w grach takich jak Go czy szachy [4].

1.3. Cel i zakres pracy

Celem pracy inżynierskiej jest zaprojektowanie i implementacja cyfrowej wersji gry karcianej, inspirowanej popularną grą Uno, z zastosowaniem algorytmów Monte Carlo Tree Search (MCTS) i strategii opisanej poniżej oraz multiplayerem umożliwiającym grę dla dwóch graczy, za pomocą internetu. Praca ta, ma na celu nie tylko zaprojektowanie i implementację

samej gry, ale również zbadanie efektywności i skuteczności dwóch algorytmów w kontekście gry karcianej.

2. Sztuczna inteligencja w grach komputerowych

2.1. Zastosowanie algorytmów AI w grach

Stosowanie algorytmów sztucznej inteligencji (AI) w grach ma swoje uzasadnienie w kilku kluczowych aspektach, przyczyniających się do zwiększenia ich wartości dla potencjalnego konsumenta:

Poprawa doświadczenia gracza - AI umożliwia tworzenie bardziej zaawansowanych i przekonujących przeciwników sterowanych przez komputer. Pozwala to nie tylko na udoskonalenie umiejętności gracza, ale również wspomaga rozwój jego kreatywności, cierpliwości, koncentracji oraz myślenia strategicznego.

Symulacja realistycznych zachowań - W grach wymagających większego pokładu myślenia, takich jak *szachy*, czy grach opartych na strategiach wojennych (np. *World of Tanks*, *World of Warships*), AI pozwala na symulowanie zachowań i reakcji, które odpowiadają ludzkim.

Dostosowanie poziomu trudności - Algorytmy AI mogą automatycznie dostosowywać poziom trudności gry w zależności od umiejętności i stylu gry użytkownika. Dzięki temu gracze mogą doświadczać bardziej dynamicznej i wymagającej rozgrywki, która lepiej odpowiada ich indywidualnym potrzebom.

Generowanie nowych elementów - AI może tworzyć nieprzewidywalne i różnorodne scenariusze gry, co prowadzi do unikalnych i często zaskakujących doświadczeń dla graczy.

Rozwój technologiczny i badawczy - Gry stanowią znakomite pole do eksperymentowania i doskonalenia algorytmów AI. Przetestowanie AI w kontrolowanych, ale złożonych środowiskach gier pozwala na zdobywanie cennej wiedzy i doświadczeń.

Automatyzacja i tworzenie zawartości - AI może być wykorzystane do automatyzowania różnych aspektów produkcji gier, takich jak generowanie poziomów i map, czy tworzenie fabuły, co pozwala twórcom skupić się na innych, kreatywnych aspektach procesu tworzenia gier.

Każdy z tych punktów przyczynia się do tworzenia gier, które są bardziej: angażujące, wymagające i różnorodne, co przekłada się na lepsze doświadczenia dla użytkowników, a także rozwój całej branży gier.

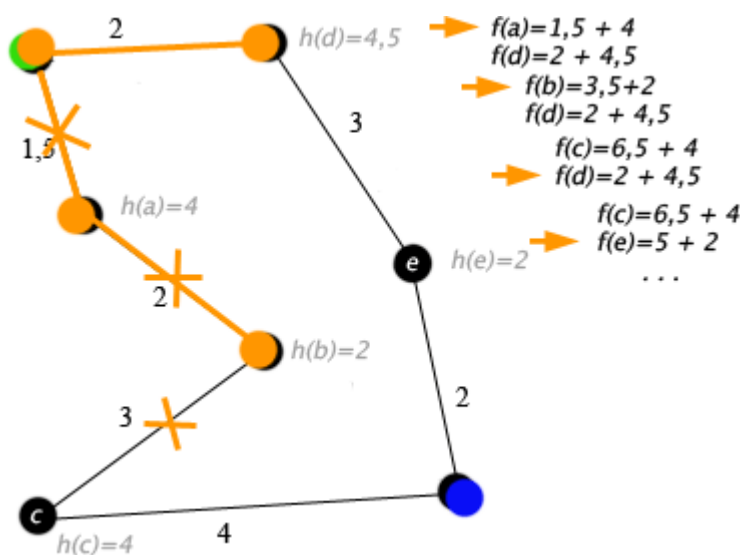
2.2. Przykładowe algorytmy

2.2.1. Algorytm A*

Algorytm A* (A-gwiazdka) jest uważany za najbardziej efektywny i powszechnie stosowanym algorytm do wyszukiwania ścieżki między dwoma punktami, np. punktami A i B w przestrzeni dwuwymiarowej. Chociaż istnieje wiele innych metod, takich jak algorytm Dijkstry czy algorytm Monte-Carlo, używany np. w AlphaGo od Google, A* wyróżnia się swoją prostotą implementacji oraz skutecznością działania.

To co sprawia, że algorytm A* jest wyjątkowy, to jego zdolność do znajdowania ścieżki pomiędzy dwoma punktami, o ile istnieje możliwość dotarcia z punktu A do B. Przejście algorytmu A* przy zachowaniu optymalnej ścieżki jest możliwe przy wykorzystaniu odpowiedniej heurystyki. Umożliwia to znalezienie optymalnej trasy w taki sposób, że żaden inny algorytm, nie jest w stanie znaleźć lepszego rozwiązania, sprawdzając przy tym mniejszą liczbę węzłów.

Zasadniczo, algorytm A* rozpatruje planszę jako graf składający się z wierzchołków połączonych krawędziami, z których każda może mieć przypisany określony koszt. W każdym kroku, algorytm wybiera wierzchołek, do którego koszt przejścia jest najniższy, kontynuując ten proces aż do osiągnięcia punktu docelowego. To podejście jest szczególnie skuteczne w przeszukiwaniu i wyznaczaniu optymalnej trasy na złożonych grafach. Idealnie obrazuje to obrazek poniżej.



Rysunek 1. Prezentacja przykładu algorytmu A*

W początkowej fazie algorytmu A-gwiazdki (A*) do rozpatrzenia są dwie krawędzie o wartościach 1,5 i 2. Do tych wartości algorytm dodaje wartość obliczoną heurystycznie - $h(a)$ dla każdego wierzchołka. W rezultacie, rzeczywiste opcje, które algorytm ma do rozważenia to:

5,5 i 6,5. W związku z tym, wybiera wierzchołek A. Następnie, stoi przed wyborem między dwoma ścieżkami: jedną o wartości $1,5 + 2 + 2$ (suma dwóch krawędzi i heurystyki) oraz drugą o wartości $2 + 4,5$. To daje łączny koszt 5,5 kontra 6,5, gdzie wybiera wierzchołek B.

Dalej, algorytm porównuje wierzchołek D, którego koszt pozostaje na poziomie 6,5, z wierzchołkiem C o koszcie 11,5 (6,5 za krawędzie plus 4 za heurystykę). W tej sytuacji wygrywa wierzchołek D. Kolejny etap to wybór między wierzchołkiem C o koszcie 11,5 a wierzchołkiem E o koszcie 7. Tym razem wybiera E. Ostatnim krokiem jest dotarcie do punktu docelowego, co kończy proces i osiąga cel.

2.2.2. Maszyny Stanów Skończonych

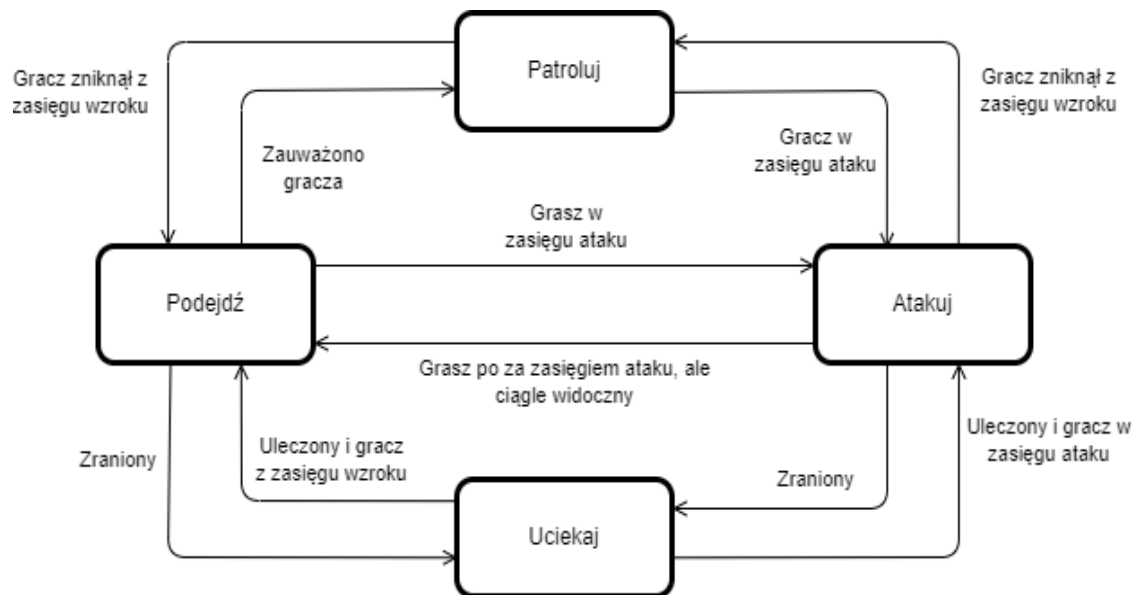
W świecie gier komputerowych, najbardziej rozpowszechnioną metodą wykorzystywaną do tworzenia sztucznej inteligencji są Maszyny Stanów Skończonych (ang. Finite-State Machine – FSM), znane również jako automaty skończone [5]. Te urządzenia działają na podstawie trzech prostych zasad:

- Automat musi posiadać określoną liczbę stanów, w których może się znajdować.
- Istnieją przejścia między tymi stanami, które są aktywowane, gdy spełnione są określone warunki.
- W danym momencie automat może znajdować się tylko w jednym stanie.

Dane wejściowe są podawane do automatu, na podstawie których określany jest stan wyjściowy. Choć może to brzmieć skomplikowanie, automat nie generuje żadnej wartości zwrotnej, a jedynie przemieszcza się z jednego stanu do drugiego. Stan wyjściowy to po prostu stan, w którym automat znajduje się po przetworzeniu danych wejściowych.

Przykłady z życia codziennego pomagają zrozumieć tę koncepcję. Weźmy np. żarówkę, która ma dwa stany: włączona i wyłączona. Innym przykładem mogą być drzwi, które mogą znajdować się w stanach: otwarte, zamknięte, zamknięte na klucz. Do manipulowania stanami drzwi, mogą być używane dwa sygnały wejściowe: „Użyj ręki” (naciśnij klamkę) oraz „Użyj klucza” (włóż i przekręć klucz w zamku).

W grach komputerowych, stany te zwykle odnoszą się do różnych akcji wykonywanych przez postacie, takich jak patrolowanie, atakowanie, ucieczka, leczenie się. Przejścia między stanami wymagają spełnienia określonych warunków, np. postać może przeładować broń tylko wtedy, gdy posiada zapasową amunicję i pusty magazynek, a atak na gracza może nastąpić tylko, gdy postać ma amunicję i widzi gracza.



Rysunek 2. Przykład zastosowania algorytmu Maszyny Stanów Skończonych

Na rysunku 2-im została zaprezentowana podstawowa wersja Maszyny Stanów Skończonych. Jest to prosty model, który można rozwijać, dodając więcej stanów i tworząc bardziej złożone warunki, np. w trakcie przejścia od podejścia do ataku, można wprowadzić warunek sprawdzający tzn. „czy gracz znajduje się w zasięgu ataku”, bazując na zmianie odległości między graczem a botem. Można też wziąć pod uwagę inne czynniki, takie jak dostępność amunicji.

Im bardziej skomplikowane są warunki i większa liczba stanów, tym bardziej rozbudowana i potencjalnie efektywniejsza staje się sztuczna inteligencja. Warunki te mogą być proste, np. reakcja na zauważenie gracza i przeprowadzenie ataku lub mogą być bardziej złożone, oparte na funkcjach, które analizują szereg różnych warunków.

Podsumowując, Maszyny Stanów Skończonych prezentują się w ten sposób i stanowią najbardziej podstawową metodę projektowania sztucznej inteligencji w grach.

2.2.3. Logika rozmyta

Zrozumienie logiki rozmytej staje się prostsze, gdy porównamy ją z logiką tradycyjną. W standardowej logice mamy do czynienia z dwoma stanami: Prawdą i Fałszem. Natomiast logika rozmyta wprowadza stan pośredni, umożliwiając operowanie na pojęciach, które nie są absolutne, lecz wymagają określenia „do jakiego stopnia” coś jest prawdziwe lub „jak bardzo” dane pojęcie się sprawdza [6]. Przykładowo, używamy określeń takich jak „bardzo małe”, „dość duże”, „średnie” i itp.



Rysunek 3. Porównanie logiki klasycznej z logiką rozmytą

Logika rozmyta umożliwia tworzenie bardziej złożonych i realistycznych systemów sztucznej inteligencji, np. poprzez wprowadzenie emocji. W takim ujęciu wirtualny przeciwnik nie jest już jednoznacznie zdefiniowany jako zły lub spokojny. Może on przyjmować różnorodne stany emocjonalne, takie jak: spokojny, podirytowany, zirytowany, bardzo zły, wściekły itd. Korzyści są podwójne. Po pierwsze, możliwe staje się operowanie na tych niedostrzegalnych stanach, a po drugie, w dialogu między programistą a projektantem gry, stwierdzenie takie jak „Przeciwnik atakuje, kiedy jest bardzo zły” nabiera większego znaczenia.

Kluczowe jest, że wartości w logice rozmytej są reprezentowane w sposób znormalizowany, czyli w skali od 0 do 1, np. 0.01, 0.23, 0.78.

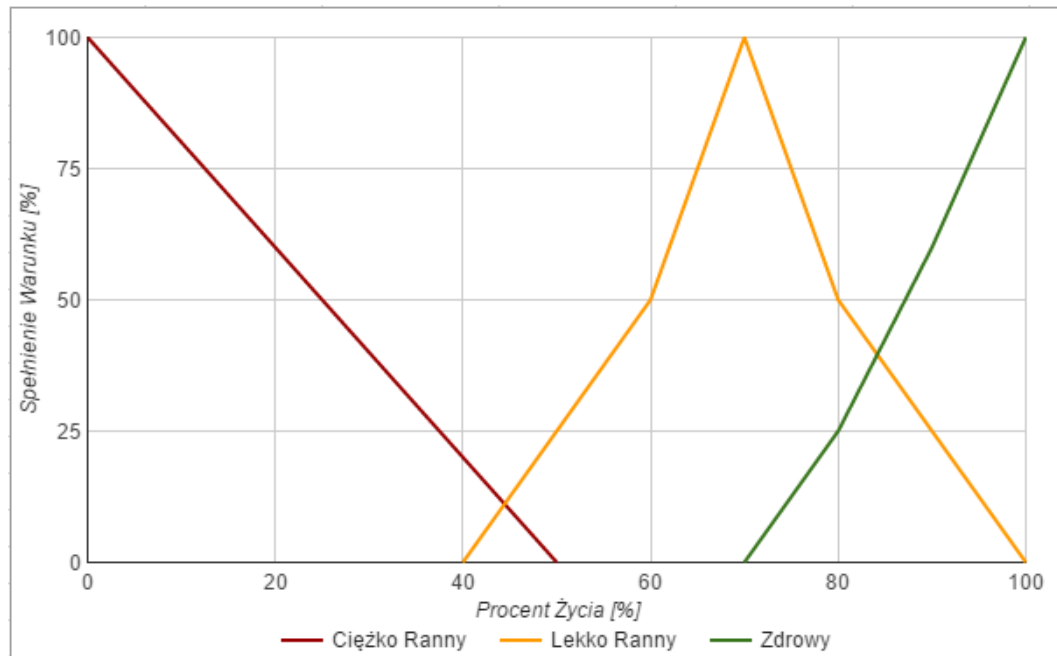
Aby efektywnie zastosować logikę rozmytą, należy najpierw określić, jakie funkcjonalności chcemy symulować. Załóżmy, że chcemy ocenić, czy bot w grze powinien użyć zaklęcia leczenia. To będzie zależało od dwóch czynników: jego poziomu bezpieczeństwa (tutaj uproszczony do odległości od gracza) oraz aktualnego stanu zdrowia.

Następnie należy zamodelować sensowne zasady działania. Może to wyglądać następująco:

- Jeśli jestem bardzo ranny i bezpieczny, rzucaj zaklęcie dużego leczenia.

- Jeśli jestem trochę ranny, rzucaj zaklęcie małego leczenia, niezależnie od poziomu bezpieczeństwa.
- Jeśli nie jestem ranny, nie używaj leczenia.
- Nie, lecz się, jeśli jest niebezpiecznie.

Teraz można określić zbiory rozmyte, które będą definiować te opisane słownie kategorie. Należy ustalić, kiedy kategoria „bardzo ranny” przechodzi w „ranny”, a „bezpiecznie” w „niebezpiecznie”. Do lepszego zwizualizowania zbiorów często używa się wykresów.

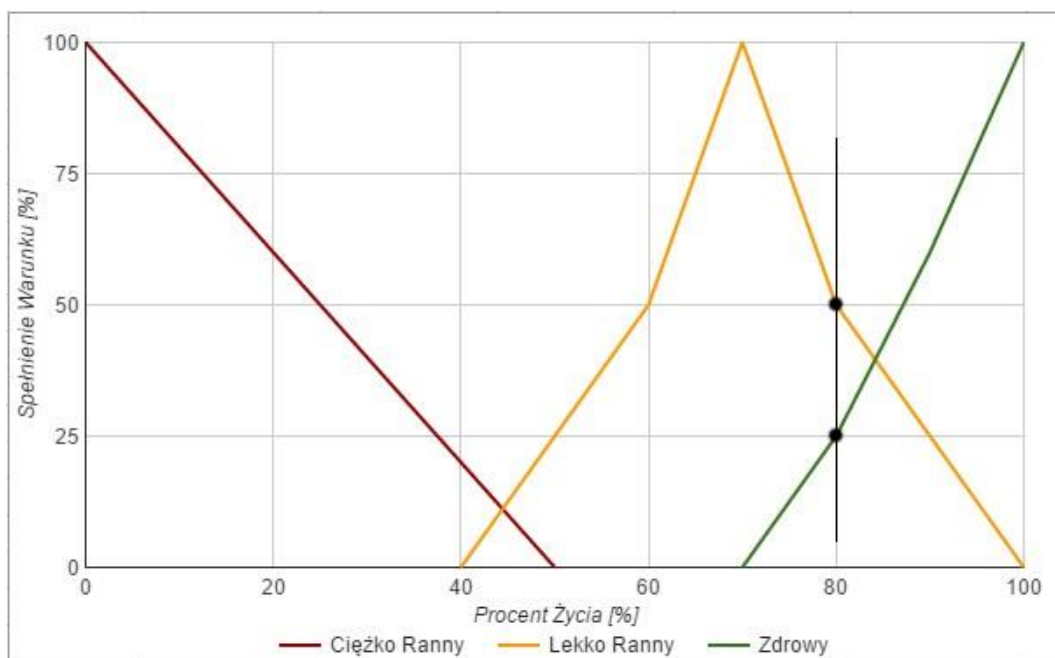


Rysunek 4. Wykres zbiorów życia bota

Rysunek 4-ty przedstawia wykres, na którym są narysowane trzy zbiory:

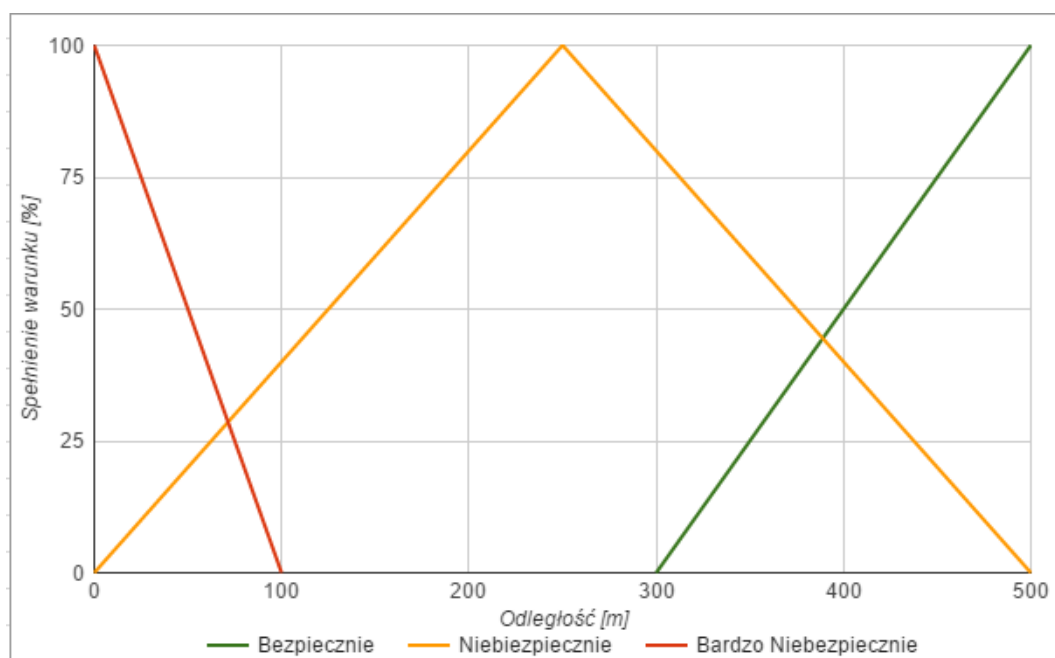
- „Ciężko Ranny” - osiąga największą wartość przy 0% życia i maleje do 0% w tym miejscu osiągając wartość 0.
- „Lekko Ranny” - zaczyna rosnąć od 40% życia bota i osiągając najwyższą wartość przy 70% życia a następnie maleje do 0 przy 100% życia.
- „Zdrowy” - zaczyna rosnąć przy 70% życia i uzyskuje maksymalną wartości przy 100% życia.

Wyobraźmy sobie sytuację, w której bot w grze posiada 80% swojego maksymalnego poziomu zdrowia.



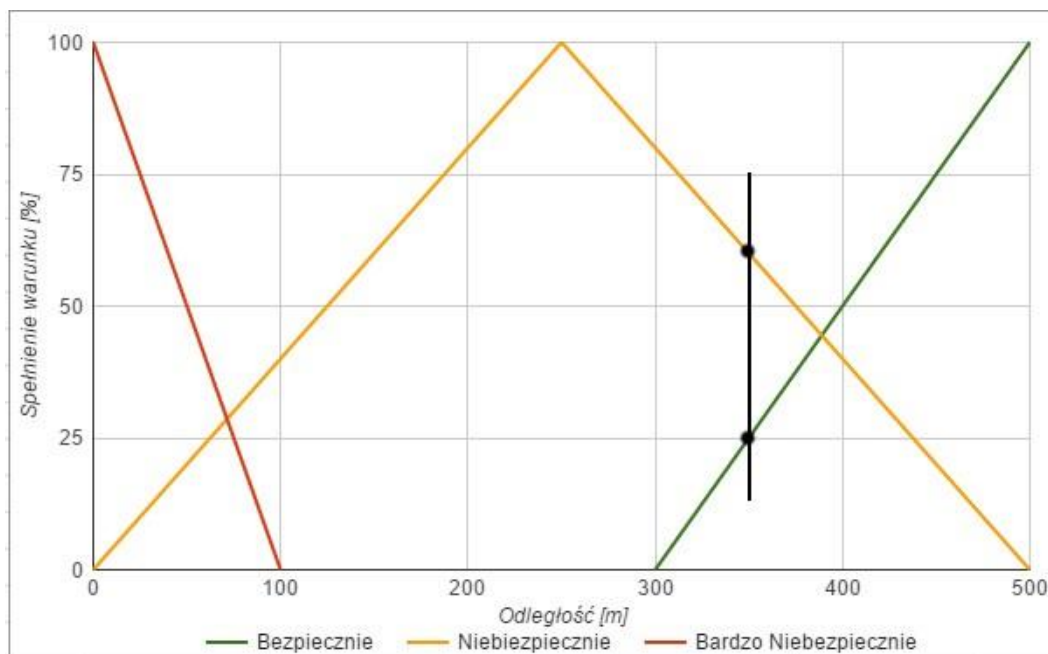
Rysunek 5. Wykres zbiorów życia bota z zaznaczonym punktem

Z tej analizy wynika, że bot jest lekko ranny z prawdopodobieństwem 0.5, a z prawdopodobieństwem 0.25 uznany jest za zdrowego. Teraz przeprowadzamy podobną ocenę dla zmiennej reprezentującej odległość. Zaczynając od utworzenia odpowiedniego wykresu



Rysunek 6. Wykres zbiorów odległości bota od gracza

Następnie, na wykresie odnoszącym się do odległości, zaznaczamy aktualną pozycję. Założmy, że w tym przypadku gracz znajduje się w odległości 350 metrów od bota.



Rysunek 7. Wykres zbiorów odległości bota od gracza z zaznaczonym punktem

Z tej sytuacji wynika, że bot znajduje się w stanie, który można uznać za 25% bezpieczny i około 65% niebezpieczny. Rozważając teraz, jakie zasady mogą być zastosowane w tej sytuacji. Gracz jest częściowo zdrowy i częściowo lekko ranny, co pozwala na zastosowanie trzech kluczowych zasad:

- Nie używam leczenia, gdy jestem zdrowy.
- Używam zakłęcia leczenia mniejszego, gdy jestem lekko ranny i sytuacja jest niebezpieczna.
- Używam zakłęcia leczenia mniejszego, gdy jestem lekko ranny i sytuacja jest bezpieczna.

Dodając do tego ocenę poziomu bezpieczeństwa, mamy możliwość zastosowania dodatkowych zasad:

- Używam zakłęcia leczenia mniejszego, gdy jestem lekko ranny i sytuacja jest niebezpieczna (co jest powtórzeniem).
- Używam zakłęcia leczenia mniejszego, gdy jestem lekko ranny i sytuacja jest bezpieczna (również powtórzenie).
- Używam zakłęcia leczenia większego, gdy jestem bardzo ranny i sytuacja jest bezpieczna.

Następnie oceniamy prawdziwość każdej z tych zasad. Prawdziwość to najniższy procent z wszystkich warunków danej zasady, np. jeśli jestem na poziomie 50% i bezpiecznie jest w 25%, to maksymalny poziom prawdziwości tej zasady to 25%. Po dokonaniu takiej ewaluacji dla każdej zasady, otrzymujemy:

- Nie używam leczenia, gdy jestem zdrowy - 25% prawdziwości (Jestem zdrowy w 25%, odległość nie ma znaczenia).
- Używam zakłęcia leczenia mniejszego, gdy jestem lekko ranny i sytuacja jest bezpieczna - 25% prawdziwości.
- Używam zakłęcia leczenia mniejszego, gdy jestem lekko ranny i sytuacja jest niebezpieczna - 50% prawdziwości.
- Używam zakłęcia leczenia większego, gdy jestem bardzo ranny i sytuacja jest bezpieczna - 0% prawdziwości.

Ostatecznie wybieramy zasadę z najwyższym stopniem prawdziwości, co w tym przypadku oznacza użycie zakłęcia leczenia mniejszego, gdyż bot jest lekko ranny i sytuacja jest niebezpieczna. W ten sposób podejmowane są decyzje w różnych sytuacjach.

3. Projekt

3.1. Architektura

Do implementacji gry wykorzystano wzorzec MVC, którego główną koncepcją jest podział aplikacji na trzy niezależne warstwy reprezentujące kolejno: (Model) Model danych - opis struktur danych i powiązań pomiędzy nimi, (View) Interfejs - czyli to, co widzi użytkownik, (Controller) Logika działania - powiązania między zdarzeniami zachodzącymi w systemie:

- View - zostały napisane z wykorzystaniem silnika szablonów Blade.
- Controller - zostały napisane przy pomocy Laravel.
- Model - zostało wykorzystane za pomocą mapowania obiektowo-relacyjnego (ORM) narzędziem Eloquent, który jest dostępny w Laravelu [7].

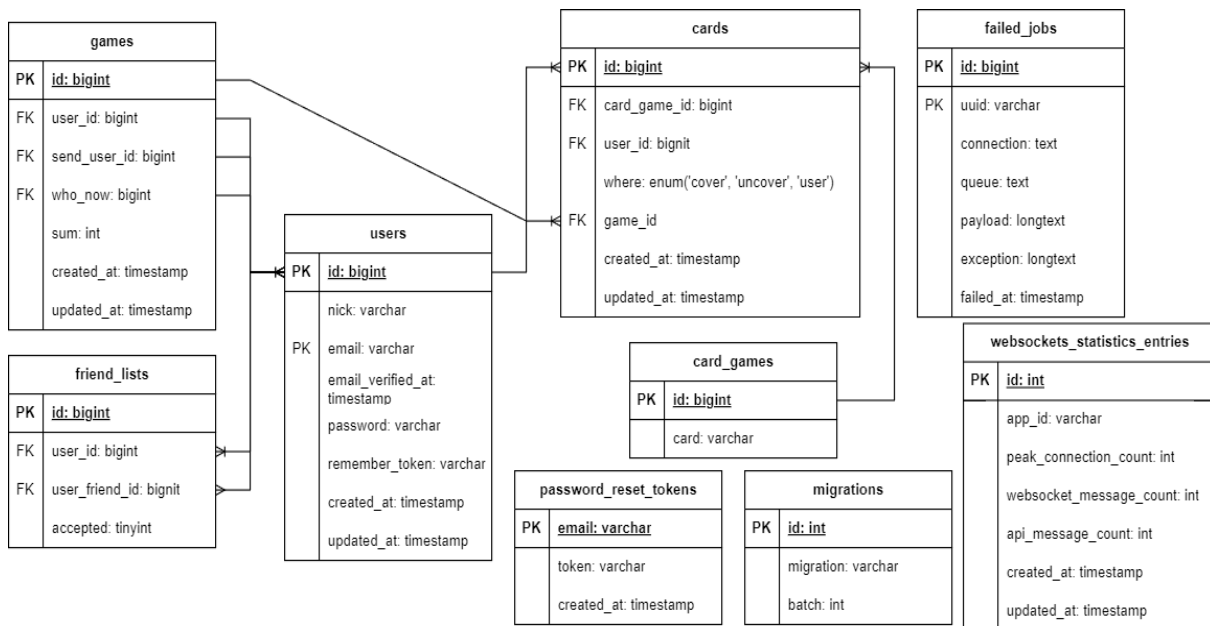
Dodatkowo do komunikacji w trybie multiplayer wykorzystano technologię WebSockets, która umożliwia otwarcie interaktywnego kanału komunikacyjnego pomiędzy przeglądarką użytkownika (klientem), a serwerem [8]. Pozwala to na dwukierunkową komunikację w czasie rzeczywistym, co oznacza, że serwer i klient mogą wysyłać dane jednocześnie, bez konieczności odświeżania strony, dzięki czemu klient może wysyłać lub odebrać dane bez utraty innych informacji, na których aktualnie wykonuje działania.

3.2. Wykorzystane technologie

- PHP - język programowania, który służy do pisania skryptów po stronie serwera [9].
- Laravel - framework PHP-owy dla aplikacji internetowych, zapewniający łatwość użycia i poprawną składnię. Używa on: wzorca MVC dla organizacji kodu, oferuje ORM Eloquent dla baz danych, posiada silnik szablonów Blade, narzędzie Artisan do automatyzacji zadań za pomocą komend, a także zaawansowane funkcje routingu i bezpieczeństwa.
- MySQL - system zarządzania bazami danych (DBMS), który używa języka zapytań SQL (Structured Query Language) do zarządzania danymi [10].
- TailwindCSS - framework CSS oparty na klasach pomocniczych, który ułatwia szybkie tworzenie niestandardowych interfejsów użytkownika [11]. Umożliwia on stosowanie małych, konkretnych klas bezpośrednio do elementów HTML.

3.3. Baza danych

3.3.1. Schemat



Rysunek 8. Schemat bazy danych

3.3.2. Opis tabel

Poniżej są opisane wszystkie tabele jakie zawiera baza danych:

websockets_statistics_entries - zawiera statystyki dla połączeń websocket. Składa się z następujących kolumn: id, app_id, peak_connection_count, websocket_message_count, api_message_count, created_at, updated_at. Tabela jest tworzona przez narzędzie WebSocKet.

users - przechowuje dane użytkowników. Takie jak: nick, email, email_verified_at (data i czas zweryfikowania adresu email), password (hasło jest zahaszkowane), rememberToken, created_at, updated_at. Tabela jest tworzona przez Laravela, gdzie zmieniono kolumnę name na nick, modyfikując ją tak, że wartości, które posiada mają być unikatowe i składające się tylko z liter i cyfr.

password_reset_tokens - tabela do zarządzania tokenami resetowania hasła, składa się z kolumn email (klucz główny), token, created_at, failed_jobs. Ta tabela jest również stworzona przez Laravela.

failed_jobs - zapisuje informacje o zadaniach, które nie zostały wykonane poprawnie. Składa się z kolumn id, uuid, connection, queue, payload, exception, failed_at. Jest automatycznie dodawana przez Laravela.

`personal_access_tokens` - tabela dla tokenów dostępu zawierająca informacje o tokenie, składa się z `id`, `tokenable`, `name`, `token`, `abilities`, `last_used_at`, `expires_at`, `created_at`, `updated_at`. Ta tabela jest również tworzona automatycznie przez Laravela.

Poniższe tabele nie zostały wygenerowane automatycznie przez używane narzędzia:

`friend_lists` - zarządza listami znajomych użytkowników. Składa się z następujących kolumn: `id` (unikalny identyfikator rekordu), `user_id` (identyfikator użytkownika, którego dotyczy lista znajomych), `user_friend_id` (identyfikator znajomego na liście) `accepted` (czy zaproszenie do znajomych zostało zaakceptowane).

`games` - zawiera informacje o grach, umożliwiając zarządzanie stanem danej gry. Składa się z następujących kolumn: `id` (unikalny identyfikator gry), `user_id` (identyfikator pierwszego użytkownika), `send_user_id` (identyfikator drugiego użytkownika), `who_now` (identyfikator użytkownika, który aktualnie może wykonać ruch), `sum` (suma kart do zabrania podczas gry jako kara), `created_at`, `updated_at` (data utworzenia i ostatniej aktualizacji rekordu).

`cards` - tabela do zarządzania kartami wszystkich gier, składa się z kart każdej gry z tabeli `games`. Jej kolumnami są: `id` (unikalny identyfikator), `card_game_id` (identyfikator nazwy karty), `user_id`: (identyfikator użytkownika posiadającego kartę, gdzie wartością domyślną jest `null`), `where` (lokalizacja, gdzie karta się znajduje, o wartościach: `cover`, `uncover`, `user`), `game_id`: (Identyfikator gry, w której używana jest karta), `created_at`, `updated_at` (daty utworzenia i ostatniej aktualizacji rekordu).

`card_games` - przechowuje nazwy wszystkich kart. Ta tabela jest używana do wyświetlania odpowiedniej nazwy karty dla użytkownika `id` (unikalny identyfikator karty), `card` (nazwa karty w grze).

4. Implementacja

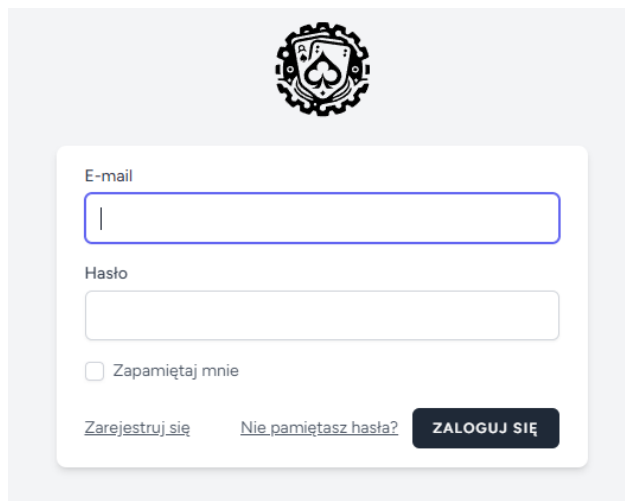
W tym rozdziale zostanie przedstawiony interfejs użytkownika opisujący każdą stronę, do której, ma dostęp użytkownik jak i również opis szczegółowy, implementacje dwóch algorytmów MCTS oraz na ustalonych zasadach pod koniec rozdziału przedstawione jest ich porównanie.

4.1. Opis interfejsu użytkownika

Poniżej jest opis całego interfejsu użytkownika, dzięki któremu może on dodawać lub usuwać ze znajomych, grać sam wybierając algorytm lub w trybie wieloosobowym, zarządzać swoim kontem, logować się, rejestrować i przypominać hasło.




Rysunek 9. Strona główna aplikacji



Rysunek 10. Strona logowania

Na rysunku 9-tym jest strona główna przedstawiająca logo aplikacji, nagłówek: „Witaj w grze Karcianka” i przycisk z napisem: „Dołącz do gry”, gdzie po kliknięciu przekierowuję użytkownika na stronę logowania.

Na rysunku 10-tym znajduje się formularz logowania z dwoma polami „E-mail” i „Hasło”, oraz trzy przyciski: „ZALOGUJ SIĘ”, który sprawdza, czy dane są poprawne i jeśli są, to przenosi do dashboradu, „Zarejestruj się”, gdzie odsyła użytkownika na stronę rejestracji oraz „Nie pamiętasz hasła?”, która odnosi na stronę z formularzem do odnowienia hasła.



Nick


E-mail

Hasło

Potwierdź Hasło

[Już zarejestrowany?](#) **ZAREJESTRUJ SIĘ**

Rysunek 11. Strona rejestracji użytkownika



Nie pamiętasz hasła? Nie ma problemu. Podaj nam swój adres e-mail, a wyślemy Ci link do zresetowania hasła, który pozwoli Ci wprowadzić nowe.

E-mail

[Logowanie](#) **WYŚLIJ LINK RESETOWANIA HASŁA**

Rysunek 12. Strona przypomnienia hasła

Rysunek po lewej przedstawia formularz na zdjęciu to standardowy formularz rejestracyjny. Zawiera następujące pola do wypełnienia:

- „Nick”: pole, w którym użytkownik wpisuje swój wymyślony nick, który musi być unikalny i składać się tylko z liter i cyfr.
- „E-mail”: pole do wpisania adresu e-mail.
- „Hasło”: pole przeznaczone do wpisania hasła, które musi mieć przynajmniej 8 znaków.
- „Potwierdź Hasło”: pole, w którym użytkownik ponownie wpisuje hasło w celu potwierdzenia.

Na dole znajduje się pytanie „Już zarejestrowany?”, gdzie po kliknięciu, użytkownik przenosi się na stronę logowania się oraz przycisk „ZAREJESTRUJ SIĘ” po prawej stronie. Jeśli użytkownik wprowadził poprawnie swoje dane - zostanie utworzone mu konto.

Na rysunku po prawej stronie jest formularz, w którym użytkownik wpisuje swój adres e-mail. Po wystaniu formularza, na podany adres e-mail wysyłany jest link, który umożliwia utworzenie nowego hasła w przypadku, gdy użytkownik zapomniał swojego obecnego hasła

Rysunek 13. Strona z formularzem do utworzenia nowego hasła

Powyższe zdjęcie przedstawia formularz poprzez, który użytkownik dostaje możliwość utworzenia nowego hasła dla swojego konta, wprowadzając nowe hasło i poniżej wpisując ponownie nowe utworzone hasło w celu potwierdzenia. E-mail jest uzupełniany automatycznie.

Rysunek 14. Strona po zalogowaniu się

Po zalogowaniu użytkownik przenosi się do dashbordu, gdzie na górze znajduje się menu z trzema zakładkami: „Panel”, „Gra” i „Znajomi”. W prawym górnym rogu jest rozwijane menu z nickiem użytkownika, pod którym znajdują się opcje: „Profil” i „Wyloguj się”, co umożliwia zarządzanie kontem lub wylogowanie się. Na środku strony znajduje się komunikat: „Jesteś zalogowany/a!”.

Rysunek 15. Strona do zaproszenia i dołączenia do gry

W zakładce „Gra” znajdują się dwie sekcje. Pierwsza sekcja umożliwia dołączenie do gry, gdzie użytkownik został zaproszony do gry. Druga sekcja służy do tworzenia gier po naciśnięciu

przycisku zagraj przy nicku znajomego. Tworzenie gier jest możliwe tylko ze znajomymi, których użytkownik posiada na swojej liście. Na końcu znajduje się przycisk „GRAJ SAM”, gdzie jest możliwość gry z wybranym algorytmem.



Rysunek 16. Strona planszy gry i historia zagranych kart

Powyższe zdjęcie przedstawia stronę z grą, która jest używana do trybu graj samemu i multiplayera. Po prawej stronie znajduje się plansza do gry w karty. Na środku planszy znajduje się zakryta karta, która umożliwia po kliknięciu dobrania kart/y zgodnie z zasadami gry. Na samej górze jest checkbox, który umożliwia pokazanie kart przeciwnika. Ta opcja jest dostępna tylko w trybie, „graj samemu”. Poniżej checkboxa są karty przeciwnika i jego nick, w trybie multiplayer jest nick naszego znajomego, a w trybie graj samemu jest napis „BOT”. Na dole planszy są karty zalogowanego gracza a powyżej jego nick. Po lewej stronie znajduje się historia zagranych kart, gdzie na samej górze znajduje się ostatnia karta, a na samym dole początkowa.

Tryby gry

☒ Heurystyczne

☐ MCTS 1000

START

Zasady gry

Wykładanie kart – podczas rozgrywki wykladać można 1 kartę. Karty na środek dokłada się według zasady koloru i figury. Jeśli więc na stole leży 5 kier, gracz może położyć dowolną 5 lub dowolnego kiera. Wyjątkiem są karty funkcyjne, które wymuszają wyłożenie na stół konkretnych figur.

Dobieranie kart – kiedy gracz nie ma w dłoni karty, którą może wyłożyć na stół, dobiera jedną z kupki i traci kolejkę. Karty dobiera też, jeśli zbiera karę. Część z kart funkcyjnych określa, ile kart powinno się dobrać. Jeśli karta karna zostaje przebita kolejną, liczba kart w karze zwiększa się. Kara kumuluje się, póki któryś z graczy nie musi dobrać karty z kupki – wtedy ponosi karę, dobierając z talii skumulowaną w karze liczbę kart.

Karty bitewne (dwójki, trójki i króle) – zmuszają gracza do dobrania tylu kart, ile na niej widnieje (Król 5 kart). Można przebijać wartości dobieranych kart tylko z kart bitewnych jeśli pasuje figura lub kolor.

Walec – Zmniejsza ilość kart do zebrania o 5.

Dama – kończy bitwę bez brania kart.

As – przenosi kart do zebrania na następnego gracza.

Karty nie funkcyjne - 4, 5, 6, 7, 8, 9, 10.

Autor grafiki kart: Designed by macrovector / Freepik

Rysunek 17. Wybór trybu gry i zasady gry

Na stronie „Graj samemu” pod stołem znajduje się formularz z dwoma opcjami, umożliwiającymi wybranie algorytmu, z którym użytkownik chce grać do wyboru: Monte Carlo Tree Search (MCTS) i drugi oparty na ustalonych zasadach. Dodatkowo przy MCTS jest ilość symulacji jakie algorytm ma przeprowadzić. Po naciśnięciu przycisku „START” uruchamia się gra z wybranym algorytmem. Na końcu strony jest lista zasad jakie są w grze.

Nataniel USUN Z LISTY ZNAJOMYCH

Maja USUN Z LISTY ZNAJOMYCH

Julia

Podaj nick znajomego DODAJ

DODAJ DO ZNAJOMYCH USUN ZAPROSZENIE

Rysunek 18. Strona do zaproszenia i dołączenia do listy znajomych

Na stronie „Znajomi” znajdują się trzy sekcje. Po lewej stronie jest lista znajomych zalogowanego użytkownika z możliwością usunięcia z grupy znajomych. Po prawej jest formularz, który po podaniu nicku znajomego umożliwia wysłanie zaproszenia do znajomych. Na dole znajduje się sekcja z otrzymanymi zaproszeniami do znajomych, gdzie możemy przyjąć lub usunąć zaproszenia.

Ostatnią stroną jest możliwość zarządzania swoim profilem. Poniższe zdjęcia prezentują wszystkie sekcje składowe.

Informacje o Profilu

Zaktualizuj informacje profilowe i adres e-mail swojego konta.

Nick

Krystian

E-mail

karol67@example.net

ZAPISZ

Rysunek 19. Pierwsza sekcja strony do zarządzania profilem

Zaktualizuj Hasło

Upewnij się, że Twoje konto używa długiego, losowego hasła, aby zachować bezpieczeństwo.

Aktualne Hasło

Nowe Hasło

Potwierdź Hasło

ZAPISZ

Rysunek 20. Druga sekcja strony do zarządzania profilem

Sekcja pierwsza umożliwia zmianę danych użytkownika, takie jak: nick i email, które muszą być unikalne. Dodatkowo przy zmianie e-maila wysyłana jest wiadomość w celu weryfikacji. Druga sekcja umożliwia zmianę hasła, w której trzeba podać aktualne hasło i nowe hasło oraz je powtórzyć.

Usuń Konto

Po usunięciu konta wszystkie jego zasoby i dane zostaną trwale usunięte. Przed usunięciem konta pobierz wszelkie dane albo informacje, które chcesz zachować.

USUŃ KONTO

Rysunek 21. Trzecia sekcja strony do zarządzania profilem

Czy na pewno chcesz usunąć swoje konto?

Po usunięciu konta wszystkie jego zasoby i dane zostaną trwale usunięte. Wprowadź hasło, aby potwierdzić, że chcesz trwale usunąć swoje konto.

Hasło

ANULUJ

USUŃ KONTO

Rysunek 22. Alert po kliknięciu usuń konto

Ostatnią sekcją jest usuwanie konta, gdzie znajduje się przycisk, po kliknięciu którego wyświetla się alert. Alert, po którym napisaniu hasła zostanie konto użytkownika usunięte i przeniesie go na stronę główną aplikacji.

4.2. Algorytm o ustalonych zasadach

4.2.1. Opis algorytmu

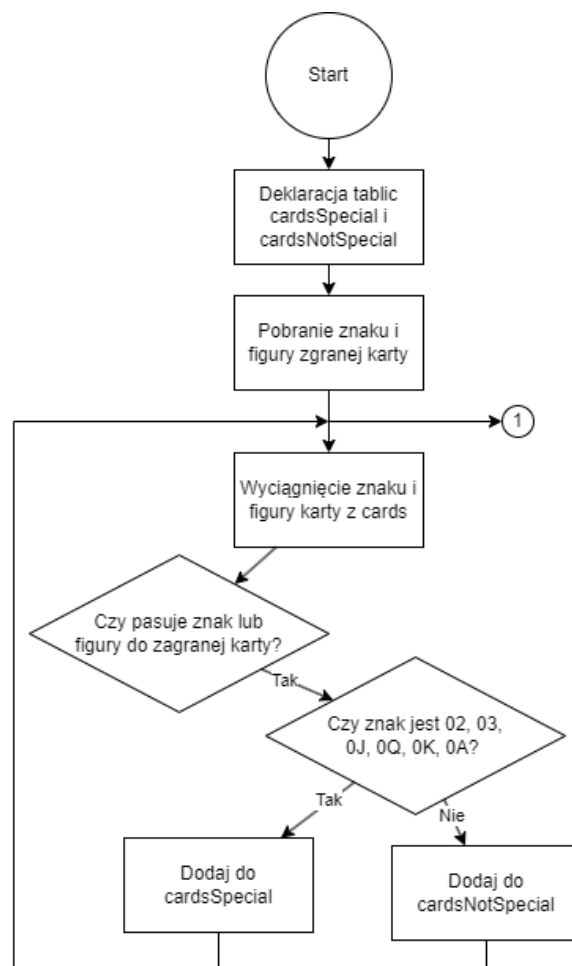
Heurystyka to sposób na rozwiązywanie problemów, który nie zawsze prowadzi do najlepszego lub nawet do poprawnego rozwiązania. Jest to rodzaj skrótu, który pomaga szybko znaleźć przybliżone rozwiązanie.

Opis przyjętych zasad do podejmowania decyzji przy wyborze karty przez algorytm:

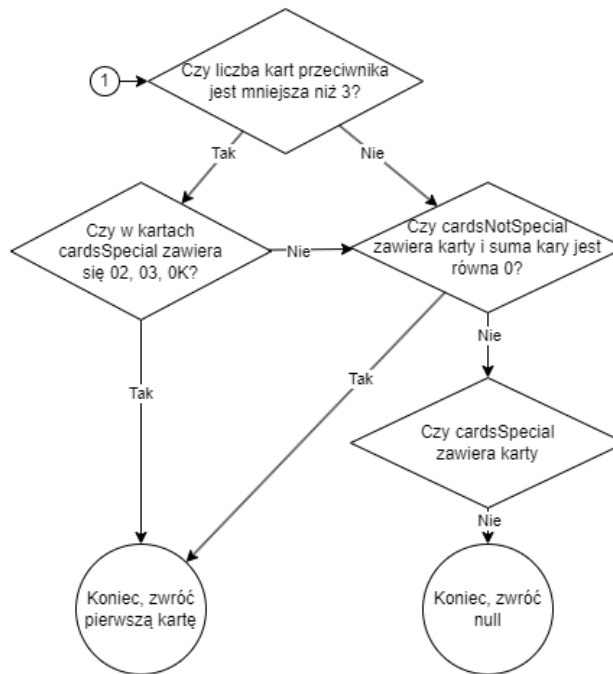
- Bot stara się pozbywać kart w swojej ręce.
- Używanie kart specjalnych w momencie, gdy musi ponieść karę.
- Bot unika używania kart specjalnych, gdy ma możliwość zagrania kartą niespecjalną.
- Jeśli przeciwnikowi zostały mniej niż trzy karty, używa karty specjalnych, która zadają karę przeciwnikowi.

4.2.2. Schemat blokowy i opis implementacji

Poniższy schemat przedstawia w jaki sposób jest podejmowanie decyzja przy wyborze karty która ma zabrać algorytm z zasad opisanych rozdział wyżej.



Rysunek 23. Pierwsza część algorytmu



Rysunek 24. Druga część algorytmu

1. Start: Początek działania algorytmu.
2. Definiuj tablice cardsSpecial i cardsNotSpecial: Tworzone są dwie puste tablice, które będą przechowywać karty specjalne i niespecjalne.
3. Pobranie znaku i figury ostatniej zagranej karty: pobierana jest wartość ostatniej zagranej karty (uncoverMainCardImg), z której wyodrębniane są znak i figura.
4. Wyciągnięcie znaku i figury kart: dla każdej karty w talii (cards) pobierana jest nazwa karty, z której wyodrębniane są znaki i figura.
5. Czy pasuje znak lub figura pod zgraną kartę? Sprawdzane jest, czy znak lub figura obecnie analizowanej karty pasują do znaku lub figury ostatniej zagranej karty.
 - a. Jeśli pasują, to algorytm przechodzi do punktu 6.
 - b. Jeśli nie pasują, to wraca do punktu 4, aby przeanalizować kolejną kartę.
6. Czy znak to: 02, 03, 0J, 0Q, 0K lub 0A? Jeśli karta pasuje, do podanych znaków to jest dodawana do tablicy cardsSpecial, a jeśli nie to do tablicy cardsNotSpecial.
7. Czy liczba kart przeciwnika jest mniejsza niż 3? Sprawdzenie czy liczba kart w posiadaniu przeciwnika jest mniejsza niż 3.
 - a. Jeśli tak, algorytm przechodzi do punktu 8.
 - b. Jeżeli nie, algorytm przechodzi do punktu 9.
8. Czy w taktach cardsSpecial jest karta z znakiem: 02, 03, 0K? Jest tu sprawdzane, czy jest tu karta z znakiem 02, 03, 0K w tabeli cardsSpecial.

- a. Jeśli tak, algorytm zwraca pierwszą kartę, która jest z znaków: 02, 03, 0K i kończy działanie.
 - b. Jeżeli nie, algorytm przechodzi do punktu 9.
- 9. Czy kara wynosi 0 i cardsNotSpecial zawiera karty? Jest tu sprawdzane, czy kara w grze jest równa 0 i tablica cardsNotSpecial zawiera karty
 - a. Jeśli tak, algorytm zwraca pierwszą kartę z tablicy cardsNotSpecial i kończy działanie.
 - b. Jeżeli nie, przechodzi do punktu 10
- 10. Czy w cardsSpecial są jakieś karty? Tu jest sprawdzanie, czy w tabeli cardsSpecial jest jakaś karta.
 - a. Jeśli tabela cardsSpecial zawiera karty, algorytm zwraca pierwszą kartę z tej tabeli.
 - b. Jeżeli tabela jest pusta, algorytm kończy działanie i zwraca wartość null, oznaczającą brak możliwości zagrania kartą.

4.3. MCTS (Monte Carlo Tree Search)

4.3.1. Opis algorytmu

Metoda MCTS, czyli Monte-Carlo Tree Search, jest techniką stosowaną w dziedzinie sztucznej inteligencji, szczególnie przydatną w określaniu najlepszych posunięć w grach komputerowych. Ta strategia skoncentrowana jest na badaniu ruchów, które wydają się najbardziej obiecujące, rozwijając drzewo możliwości, poprzez losowe wybieranie opcji z dostępnych ścieżek, co pozwala na efektywne przeszukiwanie rozległych przestrzeni decyzyjnych [12]. Metoda MCTS skupia się na analizie najbardziej obiecujących ruchów, opierając rozrost drzewa wariantów na losowym próbkowaniu przestrzeni przeszukiwań.

Sposób wyboru węzła odbywa się na podstawie wzoru wyliczający UCT (Upper ConfidenceBound 1 applied to trees), który wylicza czy bardziej opłaca się eksploatacja, czyli wykonywanie kolejnych ruchów w danej gałęzi do wygrania, czy eksploracja szukanie kolejnej gałęzi do wykonania mniej oczywistego ruchu.

$$\frac{w_i}{n_i} + c \sqrt{\frac{\ln N_i}{n_i}}$$

gdzie:

- w_i – ilość wygranych w tym węźle
- n_i – ilość wykonanych iteracji na danym węźle
- c – parametr eksploracji zazwyczaj równa $\sqrt{2}$
- N_i – jest to ilości symulacji rodzica

4.3.2. Opis implementacji

```
import {shuffleCards, special_card_check, PLAYERS} from './helper.js'
```

Kod źródłowy 1. Importowanie metod i obiektów pomocniczych

Importowanie metod i obiektów pomocniczych do algorytmu takich jak shuffleCards do tasowania kart, special_card_check do sprawdzenia warunku kart specjalnych i PLAYERS obiekt przechowujący stałe nazwy graczy.

```

export function mcts(botCards, youCards, coverMainCards, uncoverMainCards, PLAYERS, sum) {
  let initialState = new GameState(botCards, youCards, coverMainCards,
  uncoverMainCards, PLAYERS.BOT, sum)
  let mctsIteration = document.getElementById("mctsIteration").value
  let bestMove = runMCTS(initialState, mctsIteration)
  if(bestMove == "add") return null
  else return bestMove
}

```

Kod źródłowy 2. Metoda mcts

Metoda mcts jest eksportowana, którą można użyć w grze, przyjmuje następujące wartości: karty przeciwnika, użytkownika, stos kart do pobrania i już zagranych kart. Karty przeciwnika i stos kart do pobrania w głównej rozgrywce są łączone, tasowane i następnie dzielone na odpowiednie do ich poprzednich ilości. To sprawia, że algorytm nie wie jakie karty ma przeciwnik i jakie są do pobrania w następnych ruchach w oryginalnej rozgrywce.

W pierwszej linii kodu jest tworzony stan gry na podstawie klasy GameState, która zostanie opisana później. W kolejnej linii jest pobieranie ilości iteracji jakie algorytm ma wykonać. Następnie jest wykonywania metoda runMCTS, która zostanie dokładnie opisana poniżej, jej wynikiem jest karta lub wartość „add”, która mówi, że ruchem jest dobranie kart/y.

Na końcu jest zwracana karta lub wartości null do głównej gry.

```

function runMCTS(rootState, iterations) {
  const rootNode = new MCTSNode(null, null, rootState)

  for (let i = 0; i<iterations; i++) {
    let node = rootNode
    let state = rootState

    // Selection
    while (node.children.length && node.state.getPossibleMoves().length) {
      node = node.selectChild()
      state = state.makeMove(node.move)
    }

    // Expansion
    if (node.state.getPossibleMoves().length) {
      const moves = node.state.getPossibleMoves()
      const move = moves[Math.floor(Math.random() * moves.length)]
      state = state.makeMove(move)
      node = node.addChild(move, state)
    }
  }
}

```

Kod źródłowy 3. Pierwsza część metoda runMCTS

```

// Simulation
while (!state.isGameOver()) {
    const moves = state.getPossibleMoves()
    const move = moves[Math.floor(Math.random() * moves.length)]
    state = state.makeMove(move)
}

// Backpropagation
let result = state.getWinner() == PLAYERS.BOT ? 1 : 0
while (node) {
    node.update(result)
    node = node.parent
}
}

// Choose the best move at the root
return rootNode.selectChild().move
}

```

Kod źródłowy 4. Druga część metoda runMCTS

Na początku metody runMCTS jest tworzony pierwszy węzeł na podstawie klasy MCTSNode, która będzie opisana poniżej. Następnie jest wykonywana pętla z ilości iteracji jakie podał użytkownik, w której jest robiona kopia korzenia i stanu gry. W pierwszej pętli for sprawdzany jest warunek w while, czy węzeł posiada dzieci i czy są możliwe ruchy. Jeśli tak to wybierany jest nowy węzeł na podstawie wzoru podanego powyżej i robiony ruch, i zwraca nowy stan gry.

Jeśli nie, wychodzimy z pętli i następnie jest sprawdzany warunek, czy są możliwe ruchy, jeśli tak to jest losowany ruch i wykonywany on, który zwraca nowy stan gry. Następnie tworzony jest nowy węzeł dodawany jako dziecko i przepisany jako nowy węzeł, który jest zastąpiony poprzednim.

Następnie symulacja gry jest dalej wykonywana aż do końca gry. Na końcu jest sprawdzanie, kto wygrał, jeśli bot to do zmiennej rezultat jest przypisywana wartość 1, a jeśli nie - to 0. Następnie wynik jest przypisywany do węzła i wraca do rodzica węzła.

Po wykonaniu pierwszej pętli jako wynik jest zwracany ruch, najlepszego węzła wybranego z powyższego wzoru.

```

class GameState {
    constructor(botCards, humanCards, coverCards, uncoverCards, player = PLAYERS.HUMAN,
suma=0) {
        this.botCards = [...botCards]
        this.humanCards = [...humanCards]
        this.coverCards = [...coverCards]
        this.uncoverCards = [...uncoverCards]
        this.player = player
        this.suma = suma
    }

```

Kod źródłowy 5. Deklaracja klasy GameSate z konstruktorem

W konstruktorze są kopiowanie wartości do stanu gry, takie jak: karty gracza przeciwnika, do zabrania, już zagrane, kto teraz ma wykonywać ruch i suma kart jako kara.

```

    getPossibleMoves() {
        let possibleCard = []
        let unCard = this.uncoverCards.at(0)
        let uncoverCardSign = unCard.substring(0, 2)
        let uncoverCardFigure = unCard.substring(3, unCard.length)
        if(this.player == PLAYERS.BOT){
            this.botCards.forEach(card => {
                let cardSign = card.substring(0,2)
                let cardFigure = card.substring(3, card.length)
                if(cardSign==uncoverCardSign || cardFigure==uncoverCardFigure){
                    if(this.suma == 0) possibleCard.unshift(card)
                    else if(special_card_check(cardSign))
                        possibleCard.unshift(card)
                }
            })
        }
        if(this.player == PLAYERS.HUMAN){
            this.humanCards.forEach(card => {
                let cardSign = card.substring(0,2)
                let cardFigure = card.substring(3, card.length)
                if(cardSign==uncoverCardSign || cardFigure==uncoverCardFigure){
                    if(this.suma == 0) possibleCard.unshift(card)
                    else if(special_card_check(cardSign))
                        possibleCard.unshift(card)
                }
            })
        }
        if((this.suma<=this.coverCards.length && this.coverCards.length > 0) &&
possibleCard.length == 0 ) possibleCard.push("add")
        return possibleCard
    }

```

Kod źródłowy 6. Metoda getPossibleMoves w klacie GameSate

Metoda getPossibleMoves zwraca wszystkie ruchy jakie są możliwe w danym stanie gry. Na początku jest wyciągana ostanía zagrana karta i zapisana do osobnych zmiennych znak

i figura tej karty. Następnie sprawdzane jest, kto wykonuje ruch w zależności, czy jest ruch są sprawdzane karty tego gracza, czy karta pasuje pod względem znaku lub figury.

Jeśli pasuje sprawdzane jest, czy suma jest równa 0. Jeśli tak, to karta jest dodawana do możliwych ruchów. Jeśli nie, to sprawdzane jest, czy jest to karta specjalna. Jeśli tak, dodawana jest do możliwych ruchów. Na końcu, jeśli suma jest mniejsza do możliwości pobrania karty, dodawany jest „add” w celu informacji, że możliwym ruchem jest pobranie kart/y dodatkowych.

```
makeMove(move) {
  let newBotCards = [...this.botCards]
  let newHumanCards = [...this.humanCards]
  let newCoverCards = [...this.coverCards]
  let newUncoverCards = [...this.uncoverCards]
  let newSuma = this.suma
  if(this.player == PLAYERS.BOT){
    if(move == "add"){
      for(let i=1; i<newSuma; i++) newBotCards.unshift(newCoverCards.shift())
      newBotCards.unshift(newCoverCards.shift())
      newSuma = 0
    }
    else{
      newUncoverCards.unshift(move)
      newBotCards.splice(newBotCards.indexOf(move), 1)
    }
  }
  if(this.player == PLAYERS.HUMAN){
    if(move == "add"){
      for(let i=1; i<newSuma; i++){
        newHumanCards.unshift(newCoverCards.shift())
      }
      newHumanCards.unshift(newCoverCards.shift())
      newSuma = 0
    }
    else{
      newUncoverCards.unshift(move)
      newHumanCards.splice(newHumanCards.indexOf(move), 1)
    }
  }
}
```

Kod źródłowy 7. Pierwsza część metoda makeMove w klasie GameState

```

if(move != "add"){
    let sing = move.substring(0, 2)
    switch(sing){
        case "02":
            newSuma+=2
            break
        case "03":
            newSuma+=3
            break
        case "0Q":
            newSuma=0
            break
        case "0J":
            if(newSuma<=5)newSuma=0
            else newSuma-=5
            break
        case "0K":
            newSuma+=5
            break
    }
}
else{
    if(newCoverCards.length==0){
        let c = newUncoverCards.shift()
        newCoverCards = shuffleCards(newUncoverCards)
        newUncoverCards.splice(0, newUncoverCards.length)
        newUncoverCards.unshift(c)
    }
}

let newPlayer = this.player === PLAYERS.HUMAN ? PLAYERS.BOT : PLAYERS.HUMAN
return new GameState(newBotCards, newHumanCards, newCoverCards,
newUncoverCards, newPlayer, newSuma)
}

```

Kod źródłowy 8. Druga część metody makeMove w klasie GameState

W tej metodzie na początku kopiowane są wartości stanu gry. Następnie jest sprawdzenie, czyj jest ruch. Po wybraniu gracza wykonuje się ruch gracza i jeśli jest to „add”, to dodawane są mu karty zgodnie z sumą kary. Jeśli jest to co innego, to dodawana jest karta do już zagranych i graczowi usuwana. Po wykonaniu ruchu suma jest zerowania lub aktualizowana zgodnie z zasadami gry. Przed zwróceniem nowego stanu gry, sprawdzana jest konieczność dodania kart, z już zagranych kart do możliwych ich zabrania i potasowanie. Na końcu jest aktualizowane, kto teraz ma wykonywać ruch i zwracany nowy stan gry.


```

isGameOver() {
    if(this.humanCards.length == 0 || this.botCards.length == 0 || ((this.cover-
Cards.length == 0 && this.uncoverCards.length==1) || (this.coverCards.length<this.suma))){
        return true
    }
    else return false
}

```

Kod źródłowy 9. Metoda isGameOver w klasie GameState

Metoda zwraca prawdę, jeśli któryś z graczy nie ma już kart lub suma karty do zabrania jest równa 0.

```

getWinner(){
    let countHumanCards = this.humanCards.length
    let countBotCards = this.botCards.length
    if(countHumanCards == 0 || countHumanCards<countBotCards) return PLAYERS.HUMAN
    if(countBotCards == 0 || countBotCards<countHumanCards) return PLAYERS.BOT
}

```

Kod źródłowy 10. Metoda makeMove w klasie GameState

W tej metodzie zwracana jest nazwa wygranego gracza, jeśli nie ma kart lub ma mniej kart niż przeciwnik.

```

class MCTSNode {
    constructor(parent = null, move = null, state) {
        this.parent = parent
        this.move = move
        this.state = state
        this.children = []
        this.wins = 0
        this.visits = 1
    }
}

```

Kod źródłowy 11. Deklaracja klasy MCTSNode z konstruktorem

Jest to klasa reprezentująca węzeł, który jako wartości przechowuje następująco: węzeł będącym rodzicem, kartę, na której wykonano ruch, stan gry, węzły jako swoje dzieci, ilości wygranych, gdzie w następnych ruchach zakończyły się wygraną algorytmu i ilość iteracji, na którym zostały wykonane na tym węźle.

```

selectChild() {
  let selectedChild
  let bestValue = -Infinity
  this.children.forEach(child => {
    let uctValue =
      (child.wins / child.visits) +
      (Math.sqrt(2) * Math.sqrt(Math.log(this.visits) / child.visits))
    if (uctValue > bestValue) {
      selectedChild = child
      bestValue = uctValue
    }
  })
  return selectedChild
}

```

Kod źródłowy 12. Metoda selectChild z klasy MCTSNode

Metoda wylicza najlepszy węzeł z swoich dzieci na podstawie wzoru podanego wyżej.

Na końcu jest wykonywany return, który zwraca najlepszy węzeł.

```

addChild(move, state) {
  const child = new MCTSNode(this, move, state)
  this.children.push(child)
  return child
}

```

Kod źródłowy 13. Metoda addChild z klasy MCTSNode

Metoda dodaje nowy węzeł jako dziecko, na którym jest wykonywana metoda, do którego jest przypisywana karta i stan gry i zwraca nowo utworzony węzeł.

```

update(result) {
  this.visits++
  this.wins += result
}

```

Kod źródłowy 14. Metoda update z klasy MCTSNode

W ostatniej metodzie klasy MCTSNode aktualizowana jest ilość iteracji, na którym odbyły się operację na wywołanym węźle i jest przypisywany, czy udało się wygrać, czy nie.

4.4. Porównanie algorytmów

Po rozegraniu 5 gier z każdym z algorytmów okazało się, że więcej wygranych ma MCTS, ponieważ na 5 gier wygrał wszystkie, a algorytm o ustalonych zasadnych 4. To wskazywałoby, że jest nieznacznie lepszym algorytmem.

Różnice między tymi algorytmami są znaczące. Ponieważ algorytm z podejścia heurystycznego oparty na własnych zasadach jest prostszy, dzięki czemu jest szybki w implementacji i analizuje aktualny stan gry bez prowadzenia symulacji gry, dzięki czemu czas podejmowania decyzji jest szybszy od MCTS.

Natomiast trudniejszy w implementacji jest algorytm MCTS szukający jaki ruch wykonać prowadząc symulacje wybranych przez siebie gier na podstawie wzoru. Dodatkowo MCTS cechuje się większą złożonością obliczeniową co przyczynia się do czasu znalezienia rozwiązania przez ten algorytm.

5. Zakończenie

5.1. Podsumowanie i wnioski

Podsumowując wykonaną pracę, elementy takie jak: analiza tematu AI w grach komputerowych, zaprojektowanie i implementacja dwóch algorytmów heurystycznych, pierwszy oparty o ustalonych zasadach zachowania przeciwnika i MCTS (Mont-Carlo Tree Search) i implantacji multiplayer dla dwóch graczy.

Algorytm wyszukiwania drzew Monte Carlo (MCTS) to algorytm sztucznej inteligencji, którego można używać do tworzenia wymagających przeciwników w grach komputerowych. Podstawowa konstrukcja algorytmu MCTS obejmuje cztery etapy: selekcję, ekspansję, symulację i propagację wsteczną. Na etapie selekcji algorytm wybiera najbardziej obiecujący węzeł w drzewie gier w oparciu o politykę selekcji. Potem algorytm rozwija wybrany węzeł, dodając nowe węzły podrzędne, by następnie podczas symulacji, algorytm rozgrywał grę od wybranego węzła do końca. W trybie propagacji wstecznej algorytm aktualizuje statystyki węzłów w drzewie gry w dalszej części opierając się o wynik symulacji. Implementacja algorytmu MCTS może być złożona i wymagać znacznych zasobów obliczeniowych. Algorytm można zrównoleglić, aby przyspieszyć obliczenia, a zasady selekcji i symulacji można dostosować do konkretnej gry.

Algorytm MCTS ma kilka zalet, do których zalicza się możliwość znalezienia optymalnego rozwiązania w rozsądnym czasie oraz możliwość obsługi złożonych stanów gry. Jednakże algorytm może być kosztowny obliczeniowo i może nie nadawać się do zastosowań czasu rzeczywistego, takich jak gry.

Aby porównać wydajność algorytmu opartego na ustalonych zasadach i algorytmu MCTS, zostały oba algorytmy zaimplementowane w grze. Porównanie wydajności przeprowadzono poprzez pomiar współczynnika zwycięstw przeciwnika w stosunku do gracza. Przeciwnik został zaprogramowany tak, aby używał albo algorytmu podejścia heurystycznego, albo algorytmu MCTS. Wyniki porównania wydajności wykazały, że przeciwnik zaprogramowany algorytmem MCTS uzyskał lepsze wyniki niż przeciwnik zaprogramowany algorytmem na ustalonych zasadach. Można to przypisać zdolności algorytmu MCTS do znalezienia optymalnego rozwiązania i obsługi złożonych stanów gry.

5.2. Dalsze możliwości rozwoju aplikacji

Aplikacja w szerszej perspektywie daje możliwości rozwoju szeregu nowych możliwości, które znacznie wzbogacą doświadczenie użytkowników. Kluczowym elementem będzie

możliwość dodawania szczegółowych statystyk swoich gier oraz gier znajomych. Ta funkcja pozwoli graczom na śledzenie postępów, analizę strategii i porównywanie swoich osiągnięć z osiągnięciami przyjaciół, co znacząco podniesie rywalizację i zaangażowanie.

Następną funkcjonalnością jest wprowadzenie dodatkowych zasad co umożliwi użytkownikom lepsze wczucie się w grę i podniesie jakości gry. Do funkcjonalności związanej z zasadami gry, jest możliwość personalizacji zasad. Pozwoli to zwiększyć zaangażowanie użytkowników, dając im poczucie kontroli nad swoim doświadczeniem gry.

Kolejnym ważnym krokiem będzie rozbudowa opcji multiplayer. Obecnie aplikacja obsługuje gry dla dwóch graczy, rozszerzając o możliwość dodania większej liczby graczy. Dzięki temu gracze będą mogli organizować większe i bardziej dynamiczne zawody.

Wprowadzenie botów do gry multiplayer zwiększy elastyczność i dostępność rozgrywki, umożliwiając graczom cieszenie się grą w każdym momencie.

Te innowacje mają na celu nie tylko poprawę obecnych funkcji, ale także otwarcie nowych możliwości i ścieżek rozwoju dla aplikacji, co zapewni użytkownikom jeszcze bardziej satysfakcjonujące i angażujące doświadczenia.

Bibliografia

1. <https://pit.lukasiewicz.gov.pl/fakty-i-mity-na-temat-sztucznej-inteligencji-czym-jest-ai/> (26.11.23)
2. <https://trilateralresearch.com/emerging-technology/the-future-of-hobbies-how-ai-is-shaping-our-leisure-time> (28.11.23)
3. Z.J. Czech, S. Deorowicz, P. Fabian: Algorytmy i struktury danych. Wybrane zagadnienia, Wyd. Politechniki Śląskiej, Gliwice, 2010.
4. Silver D.: Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm, Cornell University, 2017.
5. A. Shukurova, B. Kumalakov, T. Zhukabayeva, S. Turaev: Modeling intelligent agents by finite machines, WFCES-II, Almaty 2021.
6. <https://geek.justjoin.it/powstaje-sztuczna-inteligencja-grach-komputerowych/> (12.10.23)
7. <https://laravel.com> (02.10.23)
8. <https://beyondco.de/docs/laravel-websockets/getting-started/introduction> (02.10.23)
9. <https://www.php.net/docs.php> (02.10.23)
10. <https://dev.mysql.com/doc/> (02.10.23)
11. <https://tailwindcss.com> (02.10.23)
12. Munos Remi: From Bandits to Monte-Carlo Tree Search. Wyd. Now Foundations and Trends, 2020.

Spis rysunków i kodów źródłowych

Spis rysunków

Rysunek 1. Prezentacja przykładu algorytmu A*	7
Rysunek 2. Przykład zastosowania algorytmu Maszyny Stanów Skończonych	9
Rysunek 3. Porównanie logiki klasycznej z logiką rozmytą	10
Rysunek 4. Wykres zbiorów życia bota	11
Rysunek 5. Wykres zbiorów życia bota z zaznaczonym punktem	12
Rysunek 6. Wykres zbiorów odległości bota od gracza	12
Rysunek 7. Wykres zbiorów odległości bota od gracza z zaznaczonym punktem	13
Rysunek 8. Schemat bazy danych	16
Rysunek 9. Strona główna aplikacji	18
Rysunek 10. Strona logowania	18
Rysunek 11. Strona rejestracji użytkownika	19
Rysunek 12. Strona przypomnienia hasła	19
Rysunek 13. Strona z formularzem do utworzenia nowego hasła	20
Rysunek 14. Strona po zalogowaniu się	20
Rysunek 15. Strona do zaproszenia i dołączenia do gry	20
Rysunek 16. Strona planszy gry i historia zagranych kart	21
Rysunek 17. Wybór trybu gry i zasady gry	22
Rysunek 18. Strona do zaproszenia i dołączenia do listy znajomych	22
Rysunek 19. Pierwsza sekcja strony do zarządzania profilem	23
Rysunek 20. Druga sekcja strony do zarządzania profilem	23
Rysunek 21. Trzecia sekcja strony do zarządzania profilem	23
Rysunek 22. Alert po kliknięciu usuń konto	23
Rysunek 23. Pierwsza część algorytmu	24
Rysunek 24. Druga część algorytmu	25

Spis kodów źródłowych

Kod źródłowy 1. Importowanie metod i obiektów pomocniczych	27
Kod źródłowy 2. Metoda mcts	28
Kod źródłowy 3. Pierwsza część metoda runMCTS	28
Kod źródłowy 4. Druga część metoda runMCTS	29

Kod źródłowy 5. Deklaracja klasy GameSate z konstruktorem	30
Kod źródłowy 6. Metoda getPossibleMoves w klacie GameSate	30
Kod źródłowy 7. Pierwsza część metoda makeMove w klasie GameState	31
Kod źródłowy 8. Druga część metody makeMove w klasie GameState	32
Kod źródłowy 9. Metoda isGameOver w klasie GameState	33
Kod źródłowy 10. Metoda makeMove w klasie GameState.....	33
Kod źródłowy 11. Deklaracja klasy MCTSNode z konstruktorem.....	33
Kod źródłowy 12. Metoda selectChild z klasy MCTSNode	34
Kod źródłowy 13. Metoda addChild z klasy MCTSNode	34
Kod źródłowy 14. Metoda update z klasy MCTSNode.....	34

OŚWIADCZENIE STUDENTA O SAMODZIELNOŚCI PRACY

Albert Chazun

Imię (imiona) i nazwisko studenta

Kolegium Nauk Przyrodniczych

Informatyka

Nazwa kierunku

117 816

Numer albumu.

1. Oświadczam, że moja praca dyplomowa pt Projekt i implementacja gry karcianej opartej o grę Uno z wykorzystaniem algorytmów heurystycznych

- 1) została przygotowana przeze mnie samodzielnie*,
- 2) nie narusza praw autorskich w rozumieniu ustawy z dnia 4 lutego 1994 roku o prawie autorskim i prawach pokrewnych (t.j. Dz.U. z 2021 r., poz. 1062) oraz dóbr osobistych chronionych prawem cywilnym,
- 3) nie zawiera danych i informacji, które uzyskałem/am w sposób niedozwolony,
- 4) nie była podstawą nadania dyplomu uczelni wyższej ani mnie, ani innej osobie.

2. Jednocześnie wyrażam zgode/ nie wyrażam zgody ** na udostępnienie mojej pracy dyplomowej do celów naukowo-badawczych z poszanowaniem przepisów ustawy o prawie autorskim i prawach pokrewnych.

Rzeszów 30.01.2024r.

Albert Chazun

(miejscowość, data)

(czytelny podpis studenta)

* uwzględniając merytoryczny wkład promotora pracy

** - niepotrzebne skreślić