Wizualizacja drzewa stanów algorytmu UCT

Plan projektu

Patryk Fijałkowski
Grzegorz Kacprowicz

6 listopada 2019

Streszczenie

Poniższy dokument zawiera ogólny zarys projektu. Aplikacja ma w zamyśle pozwalać na oglądanie i dokładną analizę rozgrywki z komputerem w jedną z dwóch gier planszowych. Istnieje możliwość łatwego rozszerzenia o kolejne gry spełniające założenia wylistowane w dokumencie. Dokument przeprowadza czytelnika przez wszystkie moduły aplikacji - zaczynając od tego odpowiedzialnego za wizualizację. Pierwszy moduł, będący najistotniejszym, będzie opierał się na usprawnionej wersji algorytmu Walkera. Opisane są również moduły odpowiedzialne za logikę zaimplementowanych gier, implementację algorytmu oraz serializowanie generowanych drzew wraz ze schematami serializacji. *Aplikacja główna*, czyli ostatni opisywany moduł, jest modułem służącym do prezentacji działania poprzednich modułów. Przedstawiony jest również schemat interfejsu użytkownika, dokładnie opisujący najistotniejsze okna aplikacji. Ostatni rozdział dokumentu opisuje i uzasadnia technologie wybrane do stworzenia aplikacji.

Historia zmian

Wersja	Data	Autor(zy)	Zmiany
1.0	3.11.2019	PF, GK	stworzenie szkicu dokumentu
1.1	4.11.2019	PF, GK	stworzenie pierwszej wersji dokumentu
1.2	5.11.2019	PF, GK	naniesienie poprawek do pierwszej wersji dokumentu
1.3	6.11.2019	PF, GK	naniesienie drobnych poprawek do poprzedniej wersji dokumentu

Spis treści

1 Architektura aplikacji

Aplikacja będzie podzielona na pięć oddzielnych modułów: *algorytm*, *serializacja*, *wizualizacja*, *gry*, które będą funkcjonować w obrębie nadrzędnego modułu - *aplikacji głównej*. Cele każdego z modułów i zadania powierzone im są przedstawione poniżej.

1.1 Wizualizacja

Moduł *wizualizacja* udostępnia funkcjonalność wizualizacji dostarczonych drzew. Użytkownik będzie miał również możliwość przybliżania, oddalania oraz poruszania się po wizualizacji. Opisana interaktywność ma na celu umożliwić dokładne zbadanie struktury drzewa oraz poszczególnych wartości w interesujących go wierzchołkach.

Aby wizualizacja była czytelna, poczyniliśmy następujące założenia:

1. Krawędzie drzewa nie mogą się przecinać.
2. Wierzchołki będą ustawione od góry w rzędach, a przynależność do rzędów będzie zależała od odległości wierzchołków od korzenia.
3. Wierzchołki mają być narysowane możliwie najwęziej.

Aby wyznaczyć układ wierzchołków na płaszczyźnie, spełniając powyższe 3 założenia, skorzystamy z usprawnionego algorytmu Walkera, który działa w czasie liniowym względem liczby wierzchołków. Algorytm, który zaimplementujemy, został opisany w pracy *Improving Walker's Algorithm to Run in Linear Time*[1].

1.2 Gry

Aplikacja będzie udostępniała 2 gry planszowe umożliwiające zademonstrowanie efektywności wizualizacji oraz algorytmu. Obie gry będą umożliwiały różne tryby rozgrywki:

- **Człowiek kontra maszyna:** decyzje jednego z graczy są podejmowane przez użytkownika, natomiast drugi gracz podejmuje decyzje najoptymalniejsze z punktu widzenia algorytmu UCT.
- **Maszyna kontra maszyna:** decyzje obojga graczy są podejmowane przez algorytm.

1.3 Algorytm

Moduł *Algorytm* jest implementacją algorytmu Monte Carlo Tree Search, korzystającą z wariantu UCT. Odpowiedzialnością tego modułu jest wyznaczanie kolejnego ruchu na podstawie dostarczonego stanu gry. Opisywany moduł będzie odpowiadał za iteracyjne tworzenie drzewa stanów i przeszukiwanie go w celu wyznaczenia najbardziej korzystnego ruchu. Użytkownik będzie miał możliwość zmiany liczby iteracji algorytmu albo ograniczenie czasowe jego działania.

Aby gra była poprawnie obsłużona przez moduł *algorytm*, musi spełniać następujące założenia:

1. Rozgrywka jest prowadzona naprzemiennie przez dwóch graczy.
2. Każdy ruch ma jednoznaczny wpływ na dalszą rozgrywkę (rozgrywka jest deterministyczna).
3. Każdy z graczy ma dostęp do pełnej informacji o aktualnym stanie gry.

Rozdział trzeci zawiera dokładniejszy opis funkcjonalności, które należy zapewnić, by moduł *Algorytm* mógł wyznaczać kolejne ruchy danej gry.

[1] *Improving Walker's Algorithm to Run in Linear Time* - Christop Buchheim, Michael Jünger, Sebastian Leipert, Universität zu Köln, Institut für Informatik, 2002

Algorytm jest opisany w formie pseudokodu w listingu 1. Ponadto, w komentarzach zamieszczone są odwołania do metod wyszczególnionych na diagramie z rysunku 1. Nasza implementacja będzie oparta o pracę *Bandit based Monte-Carlo Planning*[2].

Algorytm UCT, będący wariantem Monte Carlo Tree Search, opiera się na rozbudowywaniu drzewa ze stanami gry, poprzez iteracyjne wykonywanie czterech faz, opisanych poniżej.

1. **Faza wyboru** (linia 6 w listingu) - wybranie najbardziej obiecującego wierzchołka do rozrostu drzewa. Istotny w tej fazie jest balans pomiędzy eksploracją ruchów przeanalizowanych najdokładniej oraz eksploatacją jeszcze niezbadanych.
2. **Faza rozrostu** (linia 7 w listingu) - utworzenie wierzchołków potomnych dla najbardziej obiecującego wierzchołka. Tworzone wierzchołki odpowiadają stanom możliwym do uzyskania poprzez wykonanie jednego ruchu ze stanu wierzchołka obiecującego.
3. **Faza symulacji** (linia 8 w listingu) - rozegranie losowej rozgrywki ze stanu jednego z utworzonych wierzchołków utworzonych w poprzedniej fazie.
4. **Faza propagacji wstecznej** (linia 9 w listingu) - aktualizacja informacji wierzchołków na ścieżce od wierzchołka, z którego rozpoczęto symulację, do korzenia drzewa.

```
1  def find_next_move(curr_state):
2    iterations_counter = 0
3    tree = initialize_tree(curr_state)
4
5    while iterations_counter < max_iterations_counter:
6      curr_node = select promising node based on UCT formula      # selection(tree.root)
7      create child nodes from node                                # expansion(node)
8      playout_result = simulate random playout from curr_node     # simulation(node)
9      update tree according to playout_result                     # backpropagation(node, playout_result)
10     iterations_counter++
11
12   best_state = select best child(tree.root)
13   return best_state
```

Listing 1: Pseudokod algorytmu Monte Carlo Tree Search

W listingu 1 operujemy na trzech istotnych zmiennych - `tree` , `curr_node` i `curr_state` . Odpowiedzialnością struktury opisującej `tree` jest przechowywanie korzenia drzewa oraz stanu wyjściowego rozgrywki, który jest tej samej struktury co zmienna `curr_state` . Struktura opisująca `curr_node` przechowuje wszystkie informacje na temat wierzchołka drzewa, wraz z referencjami do wierzchołków potomnych i rodzica. Dokładniejszy opis użytych struktur jest w rozdziale 2.1.

[2] *Bandit based Monte-Carlo Planning* - Levente Kocsis, Csaba Szepesvári, Berlin, Germany, September 18–22, 2006

1.4 Serializacja

Serializacja jest modułem odpowiadającym za zapisywanie drzew do plików formacie binarnym lub csv. Oba schematy są rekurencyjne, bo taka jest również struktura generowanych przez aplikację drzew. To oznacza, że w celu zapisania całego drzewa, wystarczy zserializować jego korzeń.

Serializacja binarna

W serializacji binarnej przyjmujemy opisany niżej schemat.

- **liczba całkowita** - wartość liczby zakodowanej w U2 na 4 bajtach. Bajty liczby w kolejności little endian.
- **napis**:
  - liczba bajtów w napisie *(liczba całkowita)*,
  - zawartość napisu kodowana w UTF8.
- **liczba zmiennoprzecinkowa** - wartość liczby zakodowanej w IEEE754 na 64 bitach w kolejności little endian.
- **wierzchołek:**
  - nazwa stanu *(napis)*,
  - m - liczba węzłów potomnych *(liczba całkowita)*,
  - m powtórzeń następującego bytu:
    - nazwa ruchu *(napis)*,
    - licznik odwiedzin *(liczba całkowita)*,
    - dodatkowy licznik odwiedzin *(liczba całkowita)*,
    - średnia wypłata *(liczba zmiennoprzecinkowa)*,
    - węzeł potomny *(wierzchołek)*.

Serializacja do plików csv

W serializacji do plików csv przyjmujemy, że każdy kolejny wiersz odpowiada kolejnemu wierzchołkowi drzewa, a kolejne wartości opisujące wierzchołek oddzielamy przecinkami. Ostatnią wartością jest liczba wierzchołków potomnych. Każdy wierzchołek serializujemy do wiersza postaci:

R, O, O2, W, S, D

Oznaczenia:
- R - nazwa ruchu,
- O - licznik odwiedzin,
- O2 - dodatkowy licznik odwiedzin,
- W - średnia wypłata algorytmu za ruch,
- S - nawa stanu,
- D - liczba wierzchołków potomnych.

Kolejność wierszy opisujących wierzchołki jest analogiczna do odwiedzania wierzchołków przez algorytm przeszukiwania drzewa wgłąb, począwszy od korzenia.

- Jeśli wierzchołek v ma jednego potomka v_1, to wiersz opisujący v_1 znajduje się pod wierszem opisującym v.
- Jeśli wierzchołek v ma n potomków $v_1, v_2, ..., v_n$ i żaden z potomków nie ma swoich potomków, to pod wierszem opisującym v kolejne n wierszy opisuje wierzchołki $v_1, v_2, ..., v_n$.

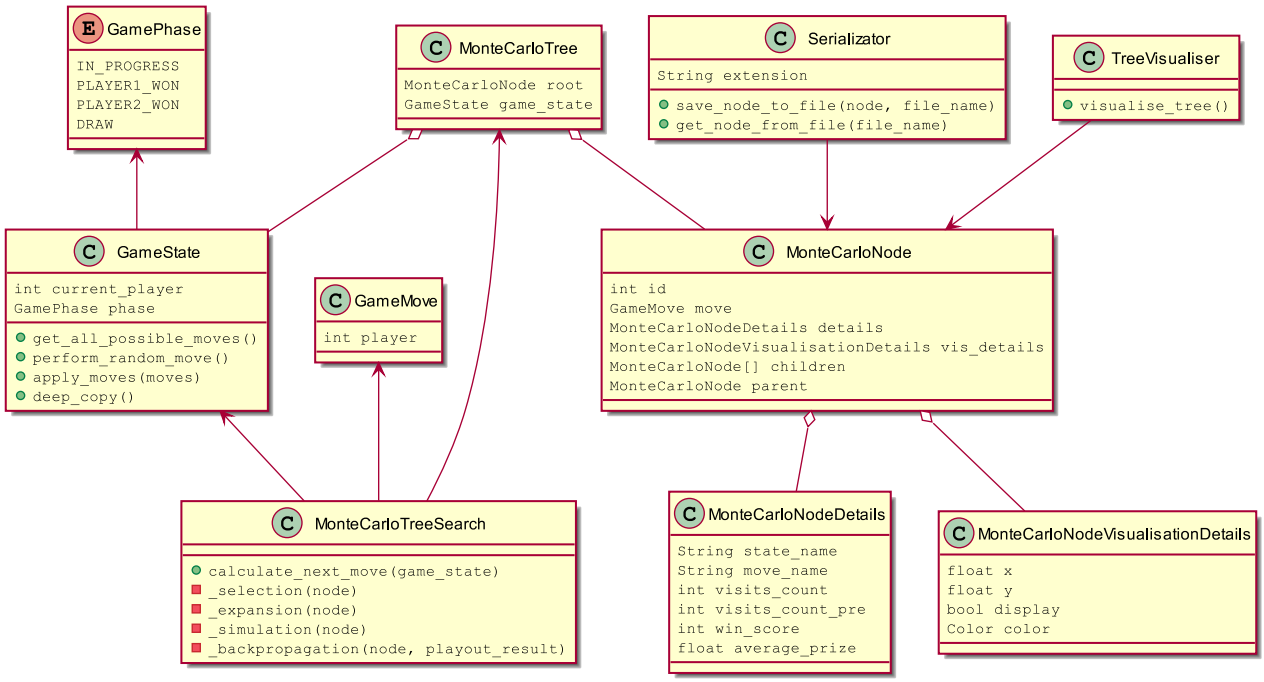
1.5 Aplikacja główna

Aplikacja główna jest modułem łączącym wszystkie pozostałe. Ten moduł skupia się na zaprezentowaniu funkcjonalności wszystkich modułów w formie aplikacji okienkowej. Obszerny opis projektu aplikacji okienkowej znajduje się w rozdziale czwartym.

2 Główne komponenty aplikacji

2.1 Diagram klas głównych komponentów

Rysunek 1 ukazuje diagram klas najważniejszych komponentów związanych z modułami *Algorytm*, *Wizualizacja* i *Serializacja*.



Rys. 1: Diagram klas głównych komponentów

Zgodnie z diagramem, klasy `MonteCarloTreeSearch`, `TreeVisualiser` oraz `Serializator` są pośrednio lub bezpośrednie zależne od klasy `MonteCarloNode`, opisującej wierzchołek w drzewie. Jest to część wspólna modułów *Algorytm*, *Wizualizacja* i *Serializacja*. Klasa `MonteCarloNode` przechowuje referencję do swojego rodzica oraz wierzchołków potomnych, aby zachować rekurencyjną strukturę drzewa.

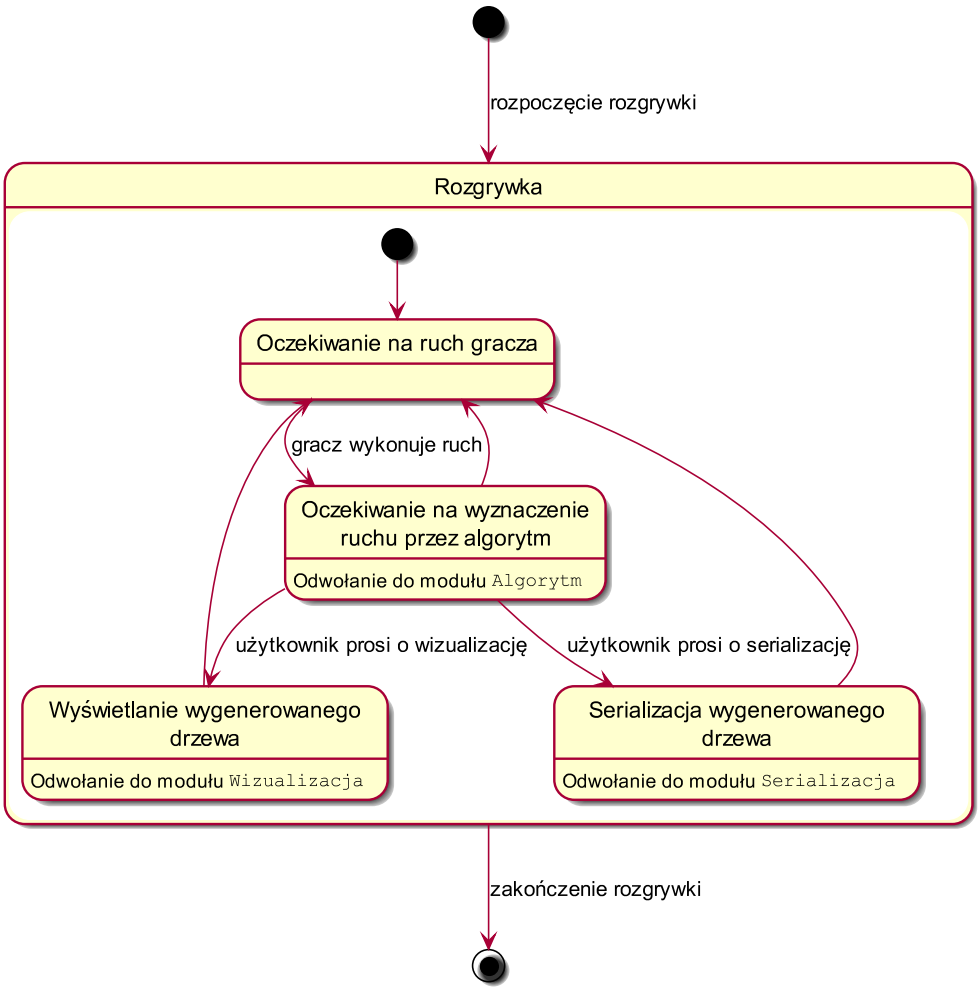
Metoda `calculate_next_move` klasy `MonteCarloTreeSearch` odpowiada za wykonanie kolejnych iteracji algorytmu. Algorytm zapisuje informacje o rozgrywanych playoutach w polach klasy `MonteCarloNodeDetails` analizowanych wierzchołków. Ruch oraz stan analizowanej gry są opisane odpowiednio przez klasy `GameMove` i `GameState`. Implementacja metod tych klas daje możliwość łatwego rozszerzenia aplikacji o inne gry. Istotny z punktu widzenia konstrukcji drzewa jest stan rozgrywki, który opisują pola typu wyliczeniowego `GamePhase`.

`TreeVisualiser` jest głównym komponentem modułu *Wizualizacja*. Jego odpowiedzialnością jest wyznaczenie układu wierzchołków drzewa na płaszczyźnie oraz wyświetlenie wygenerowanej wizualizacji. Szczegóły związane z rysowaniem każdego wierzchołka, takie jak jego współrzędne czy kolor, zawarte są w polach klasy `MonteCarloVisualisationDetails` .

`Serializator` jest klasą opisującą funkcjonalności, które mają udostępnić właściwe implementacje serializatorów, czyli serializowanie drzew do plików oraz deserializację z plików.

2.2 Diagram stanów aplikacji

Rysunek 2 ukazuje diagram stanów aplikacji w przypadku rozgrywki w trybie *człowiek kontra maszyna*.



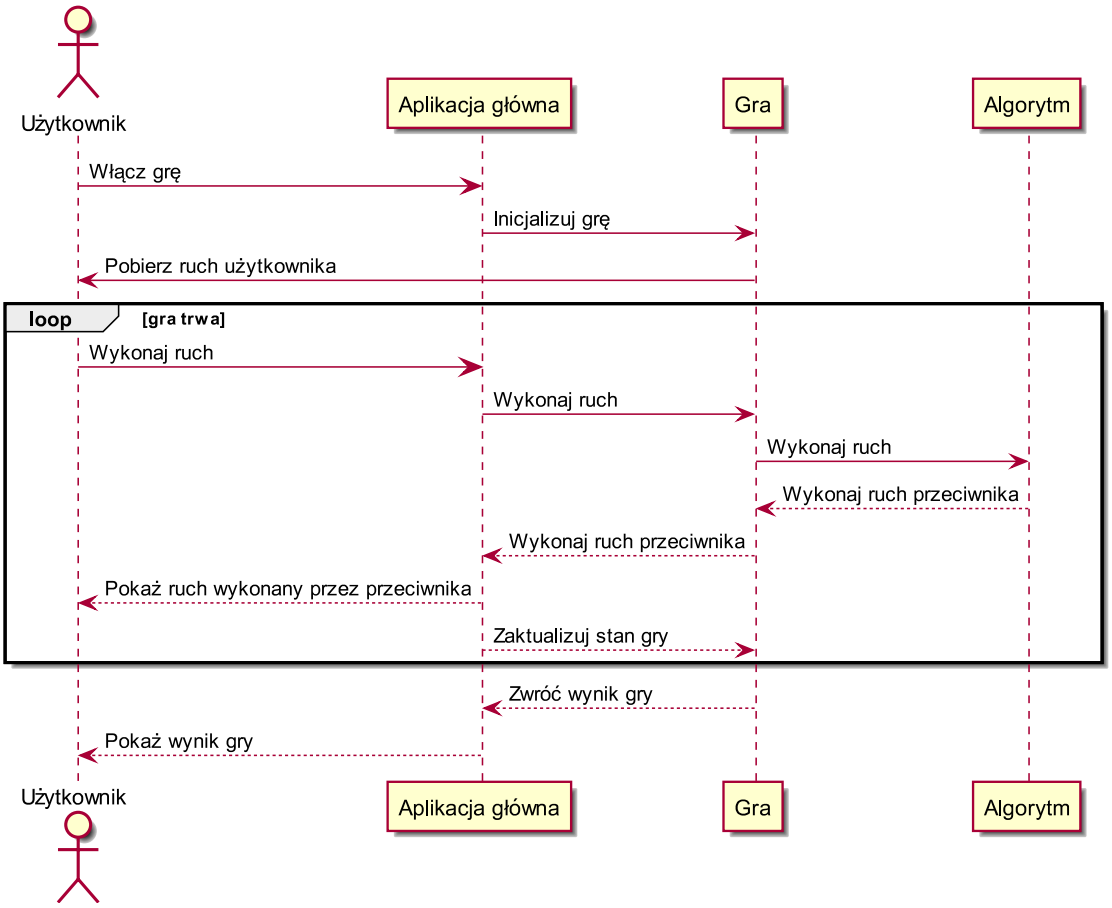
Rys. 2: Diagram stanów aplikacji

Zgodnie z diagramem, aplikacja po rozpoczęciu rozgrywki przechodzi do obszernego stanu *Rozgrywka*, zawierającego cztery wewnętrzne stany. Będac w stanie *Rozgrywka*, aplikacja może potencjalnie korzystać z każdego modułu aplikacji.

Istotna z punktu widzenia użytkownika jest możliwość serializowania wygenerowanego drzewa lub jego wizualizacja zaraz po ruchu wyznaczonym przez algorytm, co powoduje przejście aplikacji odpowiednio w stany *Serializacja wygenerowanego drzewa* oraz *Wyświetlanie wygenerowanego drzewa*.

2.3 Diagram sekwencji rozgrywki

Rysunek 3 ukazuje diagram sekwencji rozgrywki w trybie *człowiek kontra maszyna*.



Rys. 3: Diagram sekwencji rozgrywki

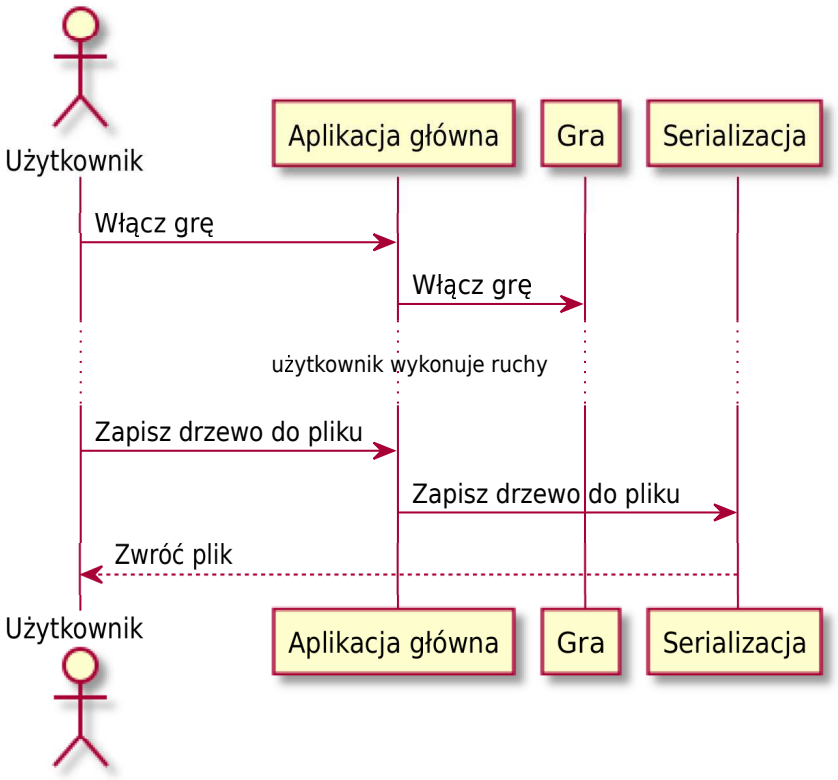
Istotne jest, jak w tej sytuacji komunikują się ze sobą moduły *Aplikacja główna*, *Gra* i *Algorytm*. Zgodnie z założeniami, *Aplikacja główna* jest interfejsem użytkownika do korzystania z pozostałych modułów.

Użytkownik końcowy za pomocą menu aplikacji głównej może ustawić parametry gry i następnie włączyć ją. Inicjalizowana jest wówczas rozgrywka w komponencie *Gra*. Następnie, dopóki gra trwa i możliwe jest wykonanie ruchu, wykonywane są na zmianę ruchy gracza i PC - wymaga to komunikacji odpowiednio użytkownika z aplikacją główną, aplikacji głównej z grą i gry z modułem *Algorytm* (i vice versa). Po zakończeniu rozgrywki gra zwraca swój stan, który jest możliwy do zobaczenia przez użytkownika poprzez okno aplikacji głównej.

Diagram ukazuje, że w tym trybie każdy ruch gracza jest ściśle związany z odpowiedzią od modułu *Algorytm*, który pobiera stan rozgrywki z modułu *Gra*.

2.4 Diagram sekwencji eksportu drzewa

Rysunek 4 przedstawia proces współpracy różnych komponentów aplikacji w celu wyeksportowania wygenerowanego przez algorytm drzewa. Proces uruchamiania gry i wykonywania ruchów jest analogiczny do tego na rysunku 3.



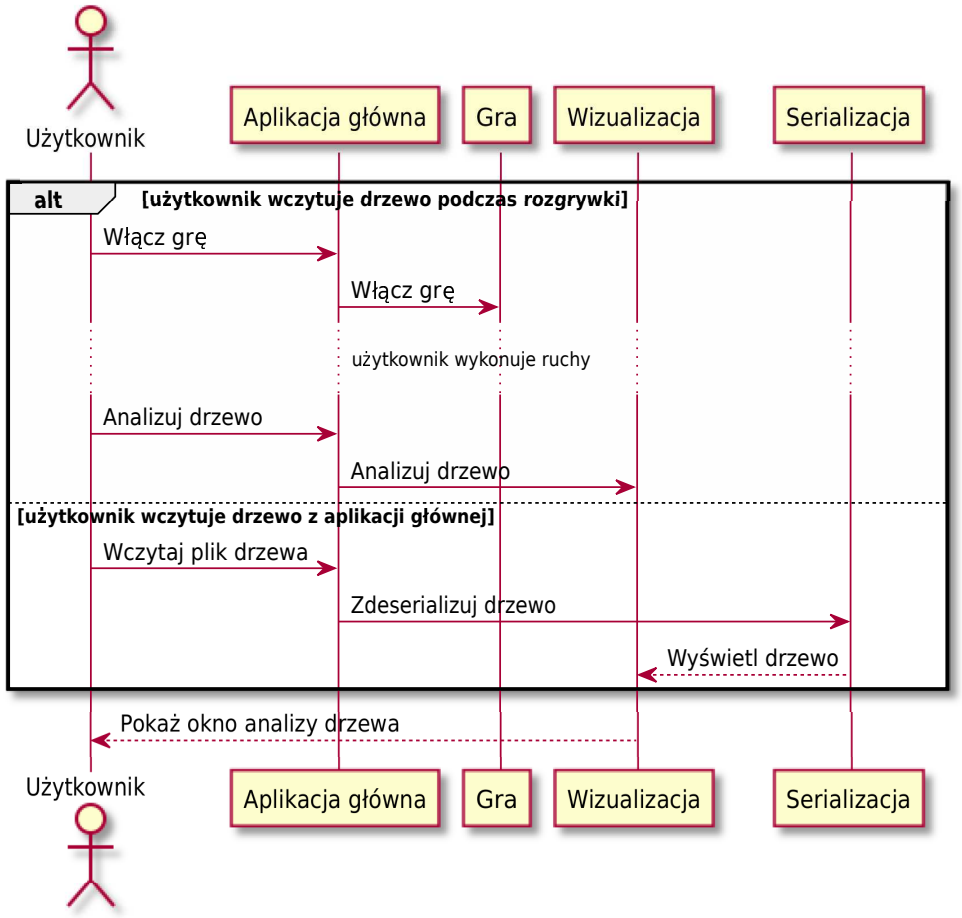
Rys. 4: Diagram sekwencji eksportu drzewa

Istotną cechą zaprojektowanego rozwiązania jest to, że gracz może wyeksportować drzewo w dowolnym momencie rozgrywki (po każdym ruchu przeciwnika). Żądanie takiej operacji przez użytkownika przesyłane jest do aplikacji głównej, która następnie komunikuje się z modułem odpowiedzialnym za serializację, który zapisuje drzewo do pliku. Plik drzewa zapisywany jest do specjalnego folderu na tego typu pliki i posiada datę wygenerowania.

Jest to diagram dla ustawienia *człowiek kontra maszyna*, jednak w przypadku *maszyna kontra maszyna* istnieje taka sama funkcjonalność i diagram byłby analogiczny.

2.5 Diagram sekwencji wizualizacji

Rysunek 5 przedstawia proces uruchamiania wizualizacji drzewa przez użytkownika jako współpracę poszczególnych komponentów aplikacji. Ponownie, proces uruchamiania gry i wykonywania ruchów wygląda tak jak na rysunku 3.



Rys. 5: Diagram sekwencji wizualizacji

Ważne jest, że użytkownik może uruchomić wizualizację z poziomu rozgrywki tuż po wygenerowaniu nowego drzewa przez algorytm lub już na etapie menu głównego. Gdy żądanie jest z poziomu rozgrywki, komponent *Aplikacja główna* komunikuje się z komponentem *Wizualizacja*, który generuje aktualne drzewo i pokazuje je użytkownikowi w nowym oknie.

Drugi sposób (żądanie analizy drzewa z menu głównego) wymaga wcześniejszego wczytania drzewa z pliku i odpowiednio jego deserializację w celu wyświetlenia - wymaga to komunikacji modułu *Wizualizacja* i *Serializacja*, gdzie ten drugi będzie zwracał wynik deserializacji temu pierwszemu. Następnie, analogicznie, użytkownik będzie mógł zobaczyć okno z wygenerowanym drzewem.

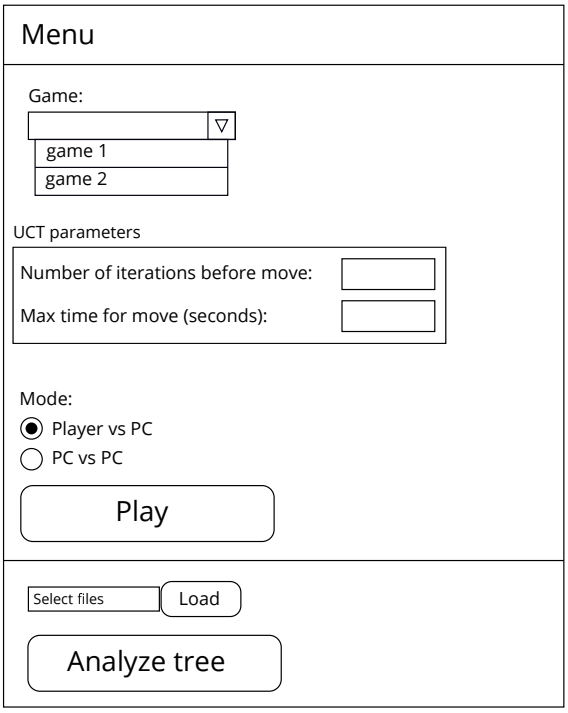
Interakcja wyżej wymienionych komponentów wygląda tak samo również w przypadku, gdy użytkownik poprosi o przeanalizowanie większej ilości drzew za jednym razem.

3 Interfejs użytkownika

Graficzny interfejs użytkownika składać się będzie z trzech głównych okien, a logika jego działania będzie w całości zawarta w module *Aplikacja główna*. Zadaniem graficznego interfejsu jest umożliwienie uruchomienia poszczególnych modułów użytkownikom końcowym.

3.1 Menu główne

Ukazane na rysunku 6 menu główne będzie głównym oknem aplikacji i będzie to pierwsza rzecz, którą zobaczy użytkownik po uruchomieniu programu.



Rys. 6: Okno menu głównego

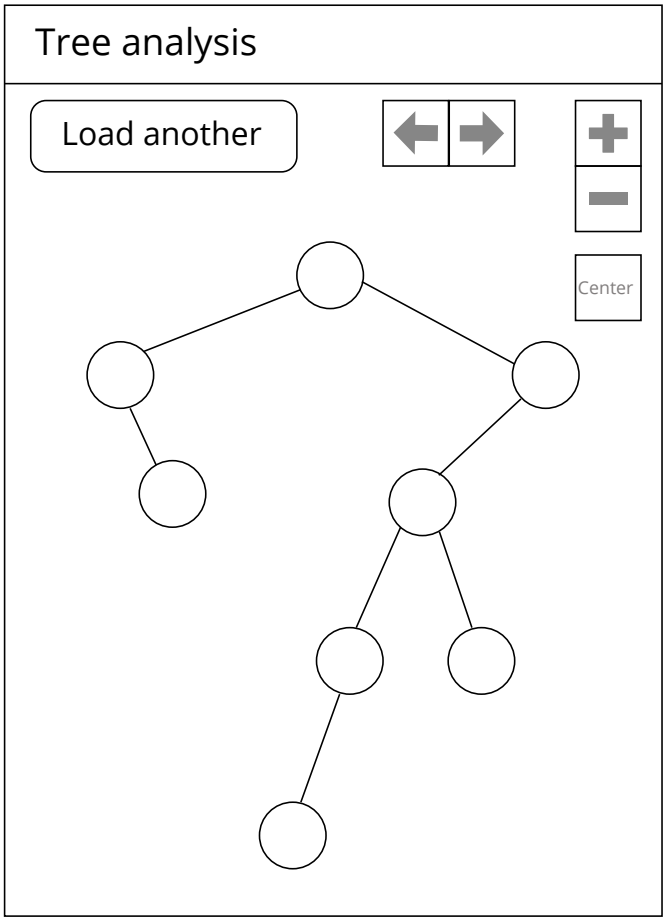
Dwa moduły, do których można przejść z tego okna, to rozgrywka i analiza drzewa. Żeby rozegrać grę, należy nacisnąć na przycisk *Play*. Powyżej tego przycisku znajdować się będzie szereg opcji, który pozwoli użytkownikowi ustawić parametry gry dostosowane do jego preferencji, w tym między innymi:

- wybór gry - rozwijana lista, w której znajdować się będą zaimplementowane gry (nasz projekt przewiduje dwa tytuły),
- liczba iteracji (rozgrywek), jaką komputer będzie wykonywał przed wykonaniem ruchu,
- maksymalny czas na wykonanie ruchu - czas, po którym komputer będzie przerywał obliczenia i wykona ruch,
- tryb rozgrywki:
  - człowiek kontra człowiek,
  - człowiek kontra maszyna,
  - maszyna kontra maszyna.

Analiza drzewa (lub drzew) będzie dostępna po naciśnięciu przycisku *Analyze tree* i uprzedniego wczytania wybranych plików (.tree, .csv).

3.2 Analiza drzewa

W oknie ukazanym na rysunku 7 będziemy mogli oglądać wczytane lub wygenerowane drzewa.



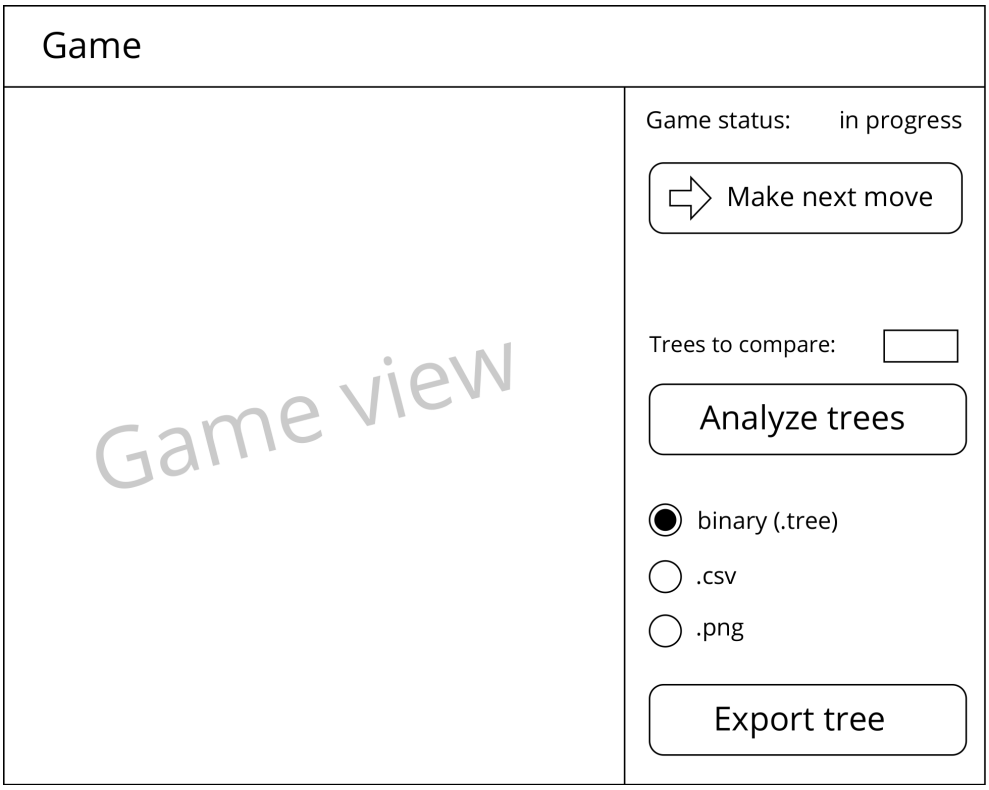
Rys. 7: Okno analizy drzewa

Kluczową funkcją będzie tutaj możliwość dynamicznego przybliżania i oddalania go (za pomocą przycisków plusa i minusa na ekranie lub scrolla) wraz z możliwością klikania poszczególnych węzłów w celu pozyskania stanu rozgrywki w danym momencie. Widoczna będzie także informacja o tym, ile razy algorytm odwiedził dany węzeł, ile razy doprowadził on do wygranej oraz średnią nagrodę za ruch w danym węźle. Możliwe będzie też wycentrowanie oglądanego drzewa.

W przypadku wczytania większej ilości drzew będzie możliwość przełączania ich za pomocą przycisków ze strzałkami w lewo i w prawo. Zaznaczane wtedy będą różnice w węzłach i krawędziach względem poprzedniego drzewa.

3.3 Rozgrywka

Poniżej przedstawiony jest przykładowy interfejs graficzny, do którego użytkownik będzie miał dostęp podczas rozgrywki.



Rys. 8: Okno rozgrywki

Zgodnie z projektem okna przedstawionym na rysunku 8, widok rozgrywki będzie podzielony na dwie części. Gra zawierać się będzie w wyżej pokazanym oknie po lewej stronie. To tutaj użytkownik za pomocą przygotowanego do gier GUI będzie mógł wykonać ruch. W prawej części okna znajdować się będą opcje związane z aktualnym stanem rozgrywki, między innymi:

- informacja o aktualnym stanie gry.
- wykonaj kolejny ruch - wyłącznie w trybie rozgrywki maszyna kontra maszyna. Użytkownik będzie miał możliwość kontrolowania wykonywanych przez komputer ruchów, aby samemu móc powodować postęp w rozgrywce.
- przeanalizuj wygenerowane drzewa - będzie to przycisk otwierający drugie okno z opisaną wcześniej analizą drzewa. Nad przyciskiem znajduje się parametr mówiący ile ruchów wstecz użytkownik ma zamiar analizować. Jedno drzewo będzie odpowiadać jednemu ruchowi komputera, a jako pierwsze wyświetli się drzewo przedstawiające stan z ostatniego ruchu i reszta będzie odpowiednio w kolejności chronologicznej (od końca).
- wyeksportuj drzewo do pliku (csv, png lub binarnego).

4 Wybrane technologie

Wybraną przez nas technologią do napisania aplikacji, to jest: gier, algorytmu i wizualizacji jest język programowania *Python* w stabilnej wersji 3.7. Do implementacji gier będziemy posługiwać się biblioteką *PyGame* (w wersji stabilnej 1.9.6). Wizualizacja będzie wykorzystywać w znacznym stopniu bibliotekę *VisPy* (OpenGL), w której najbardziej przydatną dla nas funkcją będzie możliwość pisania kodu w języku *C++* i stosunkowo łatwa integracja z głównym językiem projektu - Pythonem. Wykorzystana wersja 0.6.2 tej biblioteki również będzie wersją stabilną.

Python został przez nas wybrany ze względu na swoją wszechstronność. Posiada on bardzo szeroki zakres bibliotek, co pozwoli nam napisać zdecydowaną większość kodu w jednym języku i przyspieszyć wymianę informacji między komponentami (np. kodem gier a kodem algorytmu MCTS). Jest to korzystny scenariusz, gdyż w przeciwnym wypadku wymiana danych byłaby prawdopodobnie wolniejsza i bardziej problematyczna, ponieważ wiązałoby się to z wielokrotną serializacją i deserializacją danych.

VisPy jest nową technologią, która jest wciąż rozwijana, jednak została przez nas wybrana głównie ze względu na:
- współpracę z GPU, co będzie niezbędne podczas wizualizacji setek tysięcy wierzchołków grafu,
- obszerną dokumentację.

Wybór na PyGame padł ze względu na:
- łatwość pisania kodu i przemyślane API,
- popularność i dobrą dokumentację.