Politechnika Warszawska

WYDZIAŁ MATEMATYKI
I NAUK INFORMACYJNYCH

Praca dyplomowa inżynierska

na kierunku Informatyka

Wizualizacja drzewa stanów algorytmu UCT

Patryk Fijałkowski

Numer albumu 286350

Grzegorz Kacprowicz

Numer albumu 286358

promotor

mgr inż. Jan Karwowski

opiekun naukowy

prof. dr hab. inż. Jacek Mańdziuk

WARSZAWA 2020

....

podpis promotora podpis autora

Streszczenie

Wizualizacja drzewa stanów algorytmu UCT

Tematem pracy inżynierskiej jest implementacja projektu wizualizacji drzew stanów algorytmu z dziedziny sztucznej inteligencji - Upper Confidence Bound Applied to Trees. Dokument przedstawia wybrane zagadnienia teoretyczne i implementacyjne z zakresu tytułowego algorytmu oraz proces wytwarzania oprogramowania. Prezentowane rozwiązanie wykorzystuje ten algorytm w podejmowaniu decyzji podczas grania w dwie przykładowe gry planszowe.

Kluczową funkcjonalnością prezentowanego systemu jest przejrzysta wizualizacja ukazująca kolejne etapy rozrastania się drzew. Aplikacja jest wygodnym narzędziem do analizy działania algorytmu w czasie rzeczywistym. Dokument przeprowadza czytelnika przez wszystkie moduły aplikacji. Moduł odpowiedzialny za wizualizację, będący najistotniejszym, zawiera implementację usprawnionej wersji algorytmu Walkera. Opisane są również moduły odpowiedzialne za logikę zaimplementowanych gier, interfejs graficzny, implementację algorytmu oraz serializowanie generowanych drzew wraz ze schematami serializacji. Przedstawiony jest również obszerny opis interfejsu użytkownika, ukazujący najistotniejsze okna aplikacji. Ukazane w dokumencie instrukcje instalacji i obsługi systemu przeprowadzają czytelnika przez funkcjonalności udostępniane przez system. Finalnie, wymienione zostały konkluzje i wnioski autorów wyciągnięte z pracy nad projektem.

Słowa kluczowe: UCT, wizualizacja

Abstract

Visualization of UCT trees

The topic of this engineering diploma project is to create a system that can visualise state trees of an artificial intelligence algorithm - Upper Confidence Bound Applied to Trees. The document presents theoretical and implementation issues connected with the project and the software development process. The presented solution uses UCT algorithm to make decisions while playing two sample board games.

The key functionality of the presented system is a clear visualisation showing the subsequent stages of tree growth. The application is a convenient tool for analyzing the algorithm's operation in real time. The document guides the reader through all application modules. The module responsible for visualization, which is the most important, contains an implementation of Improved Walker's Algorithm. The modules responsible for games' logic, graphical user interface, implementation of the algorithm and the serialization of generated trees with serialization schemes are also described. A comprehensive description of graphical user interface is presented as well, showing the most important application windows. The system installation manual and user manual guide the reader through the functionalities provided by the system. Finally, the authors' conclusions drawn from the work on the project have been listed.

Keywords: UCT, visualisation

Warszawa, dnia .....

Oświadczenie

Oświadczam, że pracę inżynierską pod tytułem „Wizualizacja drzewa stanów algorytmu UCT", której promotorem jest mgr inż. Jan Karwowski, wykonałam/wykonałem samodzielnie, co poświadczam własnoręcznym podpisem.

.....

Spis treści

1. Wykaz najważniejszych oznaczeń i skrótów

- **MCTS** - Monte Carlo Tree Search
- **UCT** - Upper Confidence Bound Applied to Trees
- **CSV** - (ang. *comma-separated values*) plik w formacie .csv służący do przechowywania danych w plikach tekstowych, gdzie separatorem jest przecinek
- **PC** - (ang. *personal computer*) komputer osobisty.

2. Cel pracy

2.1. Wstęp

2.1.1. Wprowadzenie

Algorytm UCT, będący usprawnieniem metody MCTS, jest powszechnie stosowanym algorytmem w sztucznej inteligencji. Analizuje on obiecujące ruchy na podstawie generowanego drzew, równoważąc eksploatację najbardziej korzystnych z eksploracją mniej korzystnych decyzji. Każdemu wierzchołkowi drzewa odpowiada pewien stan rozgrywki, z którego algorytm rozgrywa losowe symulacje, rozszerzając potem drzewo o kolejne możliwe stany. Sposób, w jaki rozrasta się opisywane drzewo, jest kluczowy dla podejmowania przez algorytm obiecujących decyzji.

2.1.2. Cel biznesowy

Celem projektu jest stworzenie aplikacji pozwalającej na wizualizację drzewa stanów algorytmu UCT. Aplikacja będzie pozwalała na wizualizowanie drzew generowanych podczas rozgrywania dwóch przykładowych gier. Aplikacja powinna pozwalać na wizualizację drzew lub sekwencji drzew. Powinna istnieć możliwość płynnego przybliżania, oddalania i przesuwania drzewa oraz zapisu aktualnego stanu do pliku graficznego. Innymi słowy, wynikiem powinien być produkt, który pozwoliłby zrozumieć klientowi ideę i sposób działania algorytmu UCT.

2.1.3. Harmonogram pracy

Czasowy podział pracy nad projektem opisuje harmonogram z tabeli 2.1. Tabela nie opisuje zależności między kolejnymi fazami rozwoju projektu, wyznacza jedynie planowe terminy ich zakończenia. Tworząc tabelę, staraliśmy się rozłożyć pracę regularnie na cały zaplanowany czas tworzenia projektu. Ponadto, ze względu na skomplikowanie modułu odpowiedzialnego za wizualizację, wydzieliliśmy w nim trzy odrębne fazy rozwoju, wymienione poniżej.

- Podstawowa wizualizacja - możliwość wyświetlenia wszystkich wierzchołków drzewa. W podstawowej wizualizacji wygląd wierzchołków jest nieistotny.

Tablica 2.1: Harmonogram pracy

Deadline	**Przygotowane zadania**
24.10.2019	Serializacja Algorytm Pierwsza gra
7.11.2019	Podstawowa wizualizacja Połączenie algorytmu i gry
21.11.2019	Zaawansowana wizualizacja Aplikacja okienkowa Zapis drzew do pliku graficznego
5.12.2019	Pełna wizualizacja Druga gra
19.12.2019	Usprawnienia, poprawki

- Zaawansowana wizualizacja - podstawowa wizualizacja wzbogacona o możliwość analizowania statystyk poszczególnych wierzchołków. Kolor wierzchołków będzie reprezentował aktualnego gracza.
- Pełna wizualizacja - zaawansowana wizualizacja wzbogacona o możliwość przewijania, przybliżania oraz oddalania podglądu drzewa.

2.2. Założenia projektowe

2.2.1. Założenia funkcjonalne

Użytkownik korzystający z naszej aplikacji ma do wyboru jedną z dwóch gier i trzy tryby rozgrywki, wymienione poniżej.

1. Gracz versus PC – użytkownik decyduje o swoich posunięciach i mierzy się on z zaimplementowanym algorytmem.
2. PC versus PC – użytkownik jest świadkiem symulacji algorytmu, który rozgrywa partię z samym sobą.
3. Gracz versus Gracz – rozgrywka dwóch graczy, bez wizualizacji.

Zanim jednak przejdzie do rozgrywki, ma on możliwość ustawienia parametrów algorytmu, takich jak liczbę iteracji podczas tworzenia drzewa, czy też maksymalny czas na ruch przeciw-

nika. Druga opcja, którą dysponuje użytkownik, to możliwość wczytania plików reprezentujących drzewa w formacie zarówno binarnym jak i CSV, a następnie możliwość jego interaktywnej analizy. Może on wyświetlać informacje na temat wybranego węzła, a także przybliżać i oddalać całą wygenerowaną strukturę. Podczas samej rozgrywki, po wykonanym ruchu przeciwnika, gracz może analizować drzewo w sposób opisany powyżej, a także wyeksportować je. Dostępna jest możliwość zapisania go w formatach wymienionych powyżej, a także w formacie rastrowym. Użytkownik może oglądać animację rozrostu drzewa. Powyższe rzeczy dotyczą obu gier w trybach gry z udziałem algorytmu.

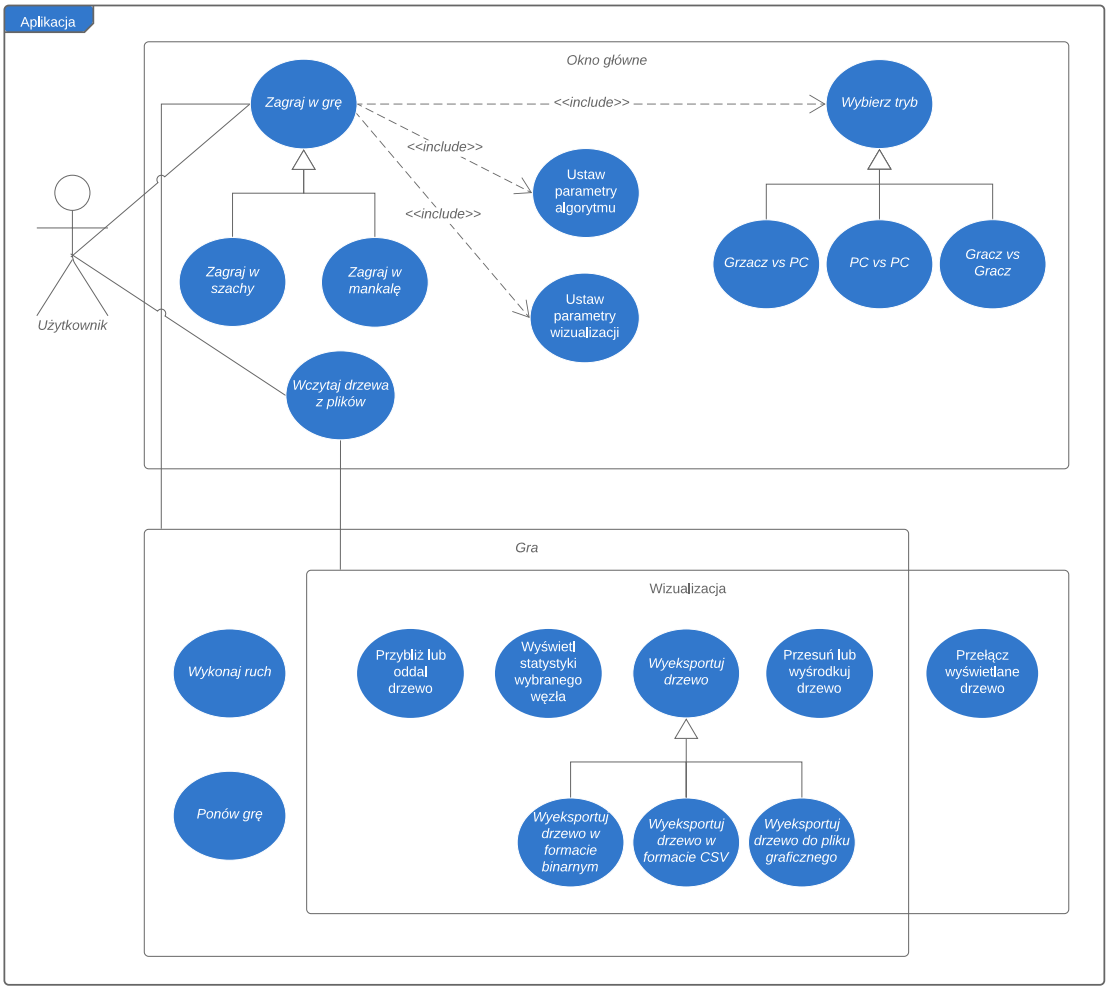
Diagram przypadków użycia, który ilustruje przedstawione możliwości, znajduje się na rysunku 2.1.

2.2.2. Założenia niefunkcjonalne

Wymagania niefunkcjonalne opisane są w tabeli 2.2 z użyciem metody FURPS. Zawiera ona opis wymagań sprzętowych aplikacji oraz jej założenia wydajnościowe.

Tablica 2.2: Analiza FURPS

Obszar	Opis
Używalność	Aplikacja działa w jednym oknie wyposażonym w przejrzysty interfejs dla użytkownika. Ponadto, dostarczona jest instrukcja instalacji i obsługi programu.
Niezawodność	Aplikacja działa na komputerze lokalnym i po zainstalowaniu jest dostępna cały czas. Potencjalne błędy nie powinny zamykać aplikacji, a jedynie wyświetlić komunikat dla użytkownika.
Wydajność	Aplikacja korzysta głównie z pamięci RAM, procesora i procesora graficznego. Dla drzew do 100 000 wierzchołków wizualizacja nie powinna zajmować więcej niż 3 sekundy, a dla 250 000 — 5 sekund. Wszystkie obliczenia i wizualizacje są przeprowadzane wewnątrz jednej maszyny.
Wsparcie	Aplikacja jest przeznaczona dla komputerów z systemami operacyjnymi Windows oraz Linux opartymi o dystrybucję Debian, na przykład Ubuntu.



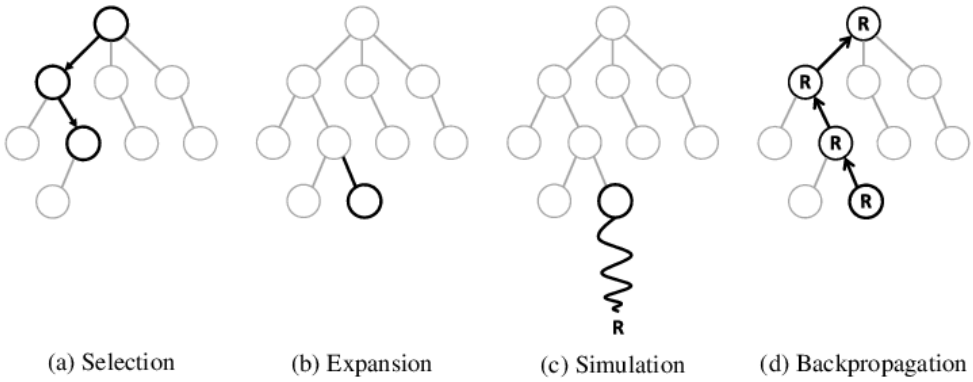
Rysunek 2.1: Diagram przypadków użycia

3. Teoria

3.1. Algorytmy MCTS

3.1.1. Opis grupy algorytmów

Monte-Carlo Tree Search to heurystyka, której celem jest podejmowanie decyzji w pewnych zadaniach sztucznej inteligencji, na przykład wybieranie ruchów w grach. Metoda jest oparta na przeszukiwaniu możliwych stanów gry zapisanych w wierzchołkach drzewa i losowym symulowaniu rozgrywek. Algorytmy MCTS opierają się na rozbudowywaniu drzewa ze stanami gry poprzez iteracyjne wykonywanie czterech faz. Jednym z najpowszechniejszych wariantów MCTS jest algorytm UCT. Pseudokod opisany w listingu 3.1 oraz implementacja MCTS w projekcie bazują na [1]. Przykład działania algorytmu ze szczególnym uwzględnieniem kolejnych faz znajduje się na rysunku 3.1.



Rysunek 3.1: Fazy MCTS

1. **Faza wyboru** (linia 7 w listingu) - wybór najbardziej obiecującego wierzchołka do rozrostu drzewa. Istotny w tej fazie jest balans pomiędzy eksploatacją ruchów przeanalizowanych najdokładniej oraz eksploracją tych jeszcze niezbadanych. Przez eksploatację wierzchołka powinno rozumieć się powiększanie rozmiaru drzewa tworzonego przez nie, a przez eksplorację — rozwijanie wierzchołków, które uprzednio nie miały żadnych potomków.

2. **Faza rozrostu** (linia 9 w listingu) - utworzenie wierzchołków potomnych dla najbardziej

obiecującego wierzchołka drzewa. Tworzone wierzchołki odpowiadają stanom możliwym do uzyskania poprzez wykonanie jednego ruchu ze stanu wierzchołka obiecującego.

3. **Faza symulacji** (linia 11 w listingu) - rozegranie partii składającej się z losowych ruchów ze stanu jednego z wierzchołków utworzonych w poprzedniej fazie. Rozgrywana jest ona do końca, czyli do wyłonienia zwycięzcy lub spowodowania remisu, lub jest ucinana po pewnej liczbie ustalonych ruchów i wynik gry jest ewaluowany przez pewną funkcję.

4. **Faza propagacji wstecznej** (linia 13 w listingu) - aktualizacja informacji na temat wierzchołków na ścieżce od wierzchołka-liścia, z którego rozpoczęto symulację, do korzenia drzewa. Główną przekazywaną wartością jest wynik symulacji.

```
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          # selection(tree.root)
7          curr_node = select promising node
8          # expansion(node)
9          create child nodes from node
10         # simulation(node)
11         playout_result = simulate random playout from curr_node
12         # backpropagation(node, playout_result)
13         update tree according to playout_result
14
15         iterations_counter++
16
17     best_state = select best child(tree.root)
18     return best_state
```

Listing 3.1: Pseudokod algorytmu Monte Carlo Tree Search

3.2. Algorytm UCT

3.2.1. Opis algorytmu

UCT jest wariantem metody MCTS, który stara się zachować równowagę między eksploatacją ruchów o wysokiej średniej wygranej a eksploracją tych mało zbadanych. Formuła, która odpowiada za wyznaczenie najbardziej obiecującego wierzchołka w fazie wyboru MCTS jest

przedstawiona jako wyrażenie 3.1.

$$\frac{w_i}{n_i} + c\sqrt{\frac{\ln N_i}{n_i}} \tag{3.1}$$

W wyrażeniu 3.1, indeks i odnosi się do liczby wykonanych przez algorytm iteracji, czyli czterech faz MCTS. W pierwszym składniku sumy wyrażenia 3.1, licznik w_i oznacza liczbę wygranych w danym węźle, a mianownik n_i oznacza liczbę rozegranych symulacji. Zatem ułamek ten przyjmuje wartości większe dla ruchów o większej średniej wygranej, co odpowiada ze eksploatację drzewa. Drugi składnik sumy wyrażenia 3.1 przyjmuje wartości większe dla wierzchołków, dla których wykonano mniej symulacji i odpowiada eksploracji drzewa. $N_i = \sum_i n_i$, a c jest parametrem eksploracji, który może być dostosowany do badanego problemu (najczęściej przyjmuje się $c = \sqrt{2}$).

3.2.2. Dodatkowe założenia

Stworzone przez nas rozwiązanie dodaje pewne założenia do algorytmu UCT. Dodanie założeń pozwoliło nam na stworzenie rozwiązania bardziej uniwersalnego i niezależnego od zasad i logiki badanych gier.

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.

3.3. Algorytm wizualizacji drzewa

3.3.1. Określenie problematyki

Celem prezentowanego rozwiązania jest wizualizacja drzewa w sposób najbardziej przejrzysty dla użytkownika. Ma to ułatwić użytkownikowi poznanie struktury drzewa i zależności między wierzchołkami. Jako że zaprojektowanie układu wierzchołków przy pewnych założeniach jest problemem NP-zupełnym nawet dla drzew binarnych, co zostało opisane w [3], w prezentowanym rozwiązaniu posłużymy się heurystyką.

3.3.2. Założenia

W celu uczynienia wizualizacji jak najbardziej czytelną, poczyniliśmy pewne założenia w kontekście układu wierzchołków i krawędzi wizualizowanych drzew. Zostały one wymienione poniżej.

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.

3.3.3. Usprawniony algorytm Walkera

W celu wyznaczenia układ wierzchołków drzewa na płaszczyźnie, spełniając powyższe 3 założenia, skorzystamy z usprawnionego algorytmu Walkera, który działa w czasie liniowym względem liczby wierzchołków. Algorytm, który zaimplementowaliśmy, został opisany w [2].

4. Implementacja

4.1. Architektura i działanie systemu

4.1.1. Wykorzystane technologie

W naszym projekcie zdecydowaliśmy się skorzystać z technologii wymienionych poniżej.

1. Języka *Python* w wersji 3.7.2, który jest udostępniany na licencji *GNU General Public License*.

2. Biblioteki *VisPy* w wersji 0.6.3, która udostępnia komponenty związane z wizualizacją graficzną. Wykorzystujemy tę bibliotekę w połączeniu z *OpenGL* w wersji 2.1. Biblioteka *VisPy* jest stworzona w oparciu o licencję *BSD*, co w kontekście projektu na pracę inżynierską pozwala na modyfikowanie i wykorzystywanie jej.

3. Biblioteki *NumPy* w wersji 1.18.1, która odpowiada za wydajne operacje na macierzach. Zgodnie z umową licencyjną opisaną przez autorów *NumPy*, można wykorzystywać ich narzędzie w zakresie pracy naukowej.

4. Nakładki na bibliotekę *Qt* - *PyQt* w wersji 5.9.2. *PyQt* umożliwia tworzenie interfejsu graficznego. Dla projektów takich jak praca inżynierska, *PyQt* dystrybuowana jest na zasadach *GNU General Public License*.

5. Biblioteki *fman build system (fbs)* w wersji 0.8.4, ułatwiającej pakowanie aplikacji korzystających z biblioteki *PyQt* w celu stworzenia pliku instalacyjnego. To oprogramowanie dystrybuowane jest na zasadach *GNU General Public License*.

6. Narzędzia *pdoc* w wersji 0.3.2, służącego do automatycznego generowania dokumentacji aplikacji. Zgodnie z umową licencyjną opisaną przez autorów *pdoc*, można wykorzystywać ich narzędzie bez ograniczeń.

4.1.2. Wstęp

Aplikacja jest podzielona na pięć oddzielnych modułów: *Algorytm*, *Serializacja*, *Wizualizacja* i *Gry*, które będą funkcjonować w obrębie nadrzędnego modułu - *Aplikacji głównej*. Cele każdego z modułów i zadania powierzone im są przedstawione w rozdziałach 4.1.3 - 4.1.10.

4.1.3. Moduł - Algorytm

Moduł *Algorytm* jest implementacją metody MCTS, 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.

W listingu 3.1, opisującym zaimplementowany algorytm, operujemy na trzech istotnych zmiennych - *tree*, *curr_node* i *curr_state*. Odpowiedzialnością struktury opisującej drzewo - *tree* - jest przechowywanie korzenia oraz stanu wyjściowego rozgrywki, który jest tej samej struktury co zmienna *curr_state*. Struktura opisująca wierzchołek – *curr_node* – przechowuje wszystkie informacje na temat wierzchołka drzewa, wraz z referencjami do wierzchołków potomnych i rodzica. Dokładne relacje między komponentami zostały opisane na diagramie 4.2.

4.1.4. Ewaluacja rozgrywek

W celu usprawnienia działania algorytmu, została dodana funkcjonalność przerywania rozgrywanych playoutów po zadanej liczbie ruchów. Po przerwaniu symulacji, w fazie propagacji wstecznej przekazana zostaje wartość nagrody wyznaczona przez ewaluację.

W przypadku mankali...

W przypadku szachów nagroda jest przyznawana w zależności od różnicy wartości figur. Zdecydowano się na użycie funkcji arctan???, której wartości w otoczeniu ??? wraz ze wzrostem argumentów gwałtownie rosną, a wraz ze spadkiem - maleją. Przekłada się to na wierniejsze odzwierciedlenie sytuacji na szachownicy, ponieważ ...

4.1.5. Moduł - Serializacja

Serializacja jest modułem odpowiadającym za zapisywanie drzew do plików formacie binarnym lub CSV. Pliki z drzewami w formacie binarnym mają umowne rozszerzenie *.tree*. Oba

schematy są rekurencyjne, bo taka jest również struktura generowanych przez aplikację drzew. To oznacza, że w celu zapisania całego drzewa, wystarczy przekazać odpowiednim komponentom jego korzeń.

4.1.6. 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 UTF-8.
- **liczba zmiennoprzecinkowa** - wartość liczby zakodowanej w IEEE754 na 64 bitach w kolejności little endian.

Schemat serializowania wierzchołka wykorzystuje wymienione powyżej zasady i ma następującą strukturę:

- 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)*.

4.1.7. 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 $v_1, v_2, ..., v_n$.

4.1.8. Moduł - Wizualizacja

Moduł *Wizualizacja* udostępnia funkcjonalność wizualizacji dostarczonych drzew. *Wizualizacja* jest jedynym modułem, który korzysta z technologii *VisPy* oraz *NumPy*. Wykorzystanie tych technologii ma na celu odpowiednio wydajne wyświetlenie wizualizacji oraz przechowywanie wektorów z danymi, które będzie można przekazać karcie graficznej. Zadania *Wizualizacji* są wymienione poniżej.

- Przypisanie wierzchołkom drzewa miejsce na płaszczyźnie przy użyciu usprawnionego algorytmu Walkera i przetransformowanie wyznaczonych koordynatów do układu współrzędnych OpenGL.
- Przypisanie krawędziom koloru w zależności od liczby odwiedzin wierzchołka.
- Przypisanie wierzchołkom koloru w zależności od gracza reprezentowanego stanu gry.
- Detekcja kliknięcia wierzchołka przez użytkownika.
- Wyświetlenie drzewa.
- Przybliżanie, oddalanie i poruszanie się po wizualizacji.

4.1.9. Moduł - Gry

Prezentowane rozwiązanie udostępnia dwie gry planszowe w ramach modułu *Gry*. System został przygotowany z myślą o rozszerzaniu o kolejne gry. Wymagania, które należy spełnić, dodając kolejną grę, są opisane w rozdziale 4.2.1.

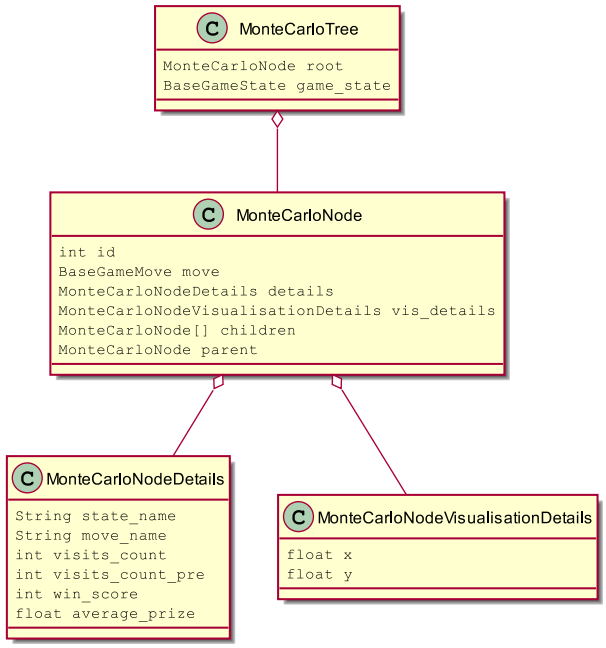
4.1.10. Moduł - Aplikacja główna

Aplikacja główna jest modułem łączącym wszystkie pozostałe. Ten moduł skupia się na zaprezentowaniu funkcjonalności wszystkich modułów w formie aplikacji okienkowej. Interfejsy, które udostępnia *Aplikacja główna*, są opisane w rozdziale 5.1.

4.2. Główne komponenty aplikacji

4.2.1. Diagram klas

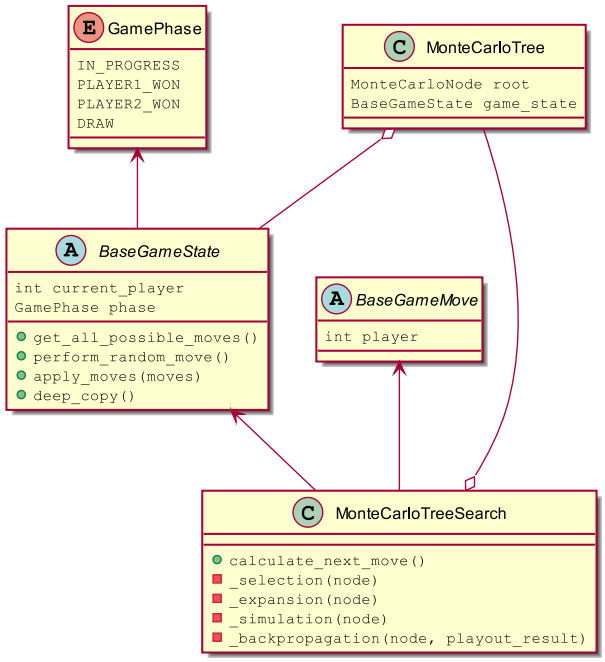
Klasa *MonteCarloNode*, zgodnie z diagramem 4.1, reprezentuje wierzchołek w drzewie, więc przechowuje referencje do swojego rodzica i wierzchołków potomnych, aby zachować rekurencyjną strukturę drzewa. Ponadto zawiera wszystkie informacje związane z przebiegiem algorytmu UCT w polu *details* oraz algorytmu Walkera w polu *vis_ details*.



Rysunek 4.1: Diagram UML klasy wierzchołka

Diagram 4.2 ukazuje najistotniejsze klasy modułu *Algorytm*. 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*.



Rysunek 4.2: Diagram UML dla modułu *Algorytm*

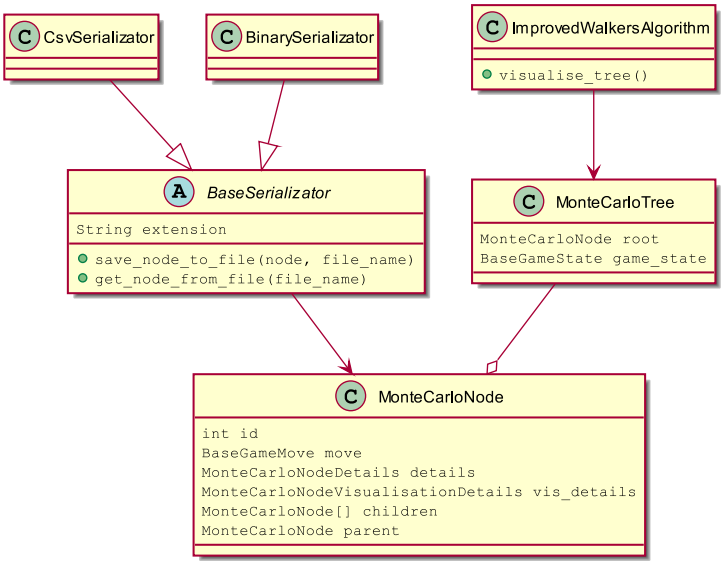
Zgodnie z diagramami 4.3 i 4.2, klasy *MonteCarloTreeSearch*, *ImprovedWalkersAlgorithm* oraz *Serializator* są pośrednio lub bezpośrednie zależne od klasy *MonteCarloNode*, opisującej wierzchołek w drzewie. Jest to część wspólna modułów *Algorytm*, *Wizualizacja* i *Serializacja*.

Widniejąca na diagramie 4.3 klasa *ImprovedWalkersAlgorithm* jest głównym komponentem modułu *Wizualizacja*. Jego odpowiedzialnością jest wyznaczenie układu wierzchołków drzewa na płaszczyźnie. Szczegóły związane z przebiegiem algorytmu Walkera i rysowaniem każdego wierzchołka, 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.

4.2.2. Diagram stanów rozgrywki

Rysunek 4.4 ukazuje diagram stanów aplikacji w przypadku rozgrywki w trybie *Gracz versus PC*.

Zgodnie z diagramem, aplikacja po zainicjowaniu rozgrywki przechodzi do stanu *Rozgrywka*



Rysunek 4.3: Diagram UML dla modułów *Serializacja* i *Wizualizacja*

zawierającego cztery stany wewnętrzne. Będąc w stanie *Rozgrywka*, aplikacja może korzysta z modułów *Algorytm* i *Wizualizacja*, a także opcjonalnie z modułu *Serializacja*.

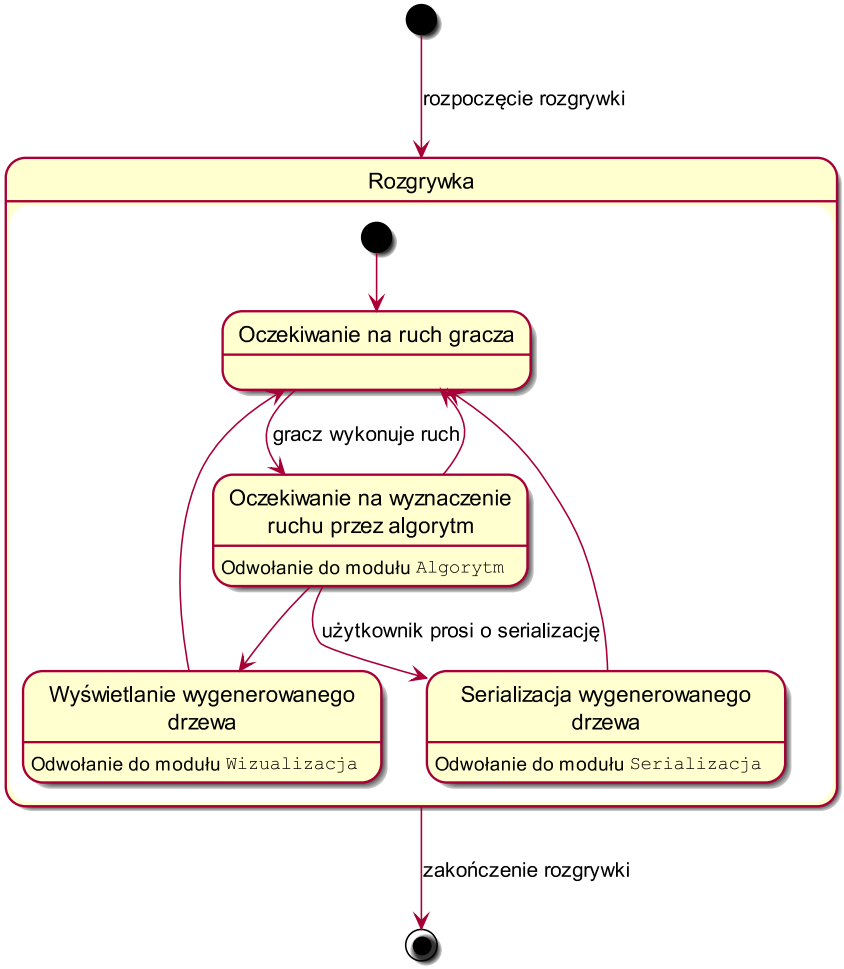
Istotna z punktu widzenia użytkownika jest wizualizacja drzewa stanów bezpośrednio po ruchu wyznaczonym przez algorytm, co powoduje przejście aplikacji odpowiednio w stany *Serializacja wygenerowanego drzewa* oraz *Wyświetlanie wygenerowanego drzewa*, a także możliwość serializacji i zapisania powstałego drzewa – stan *Serializacja wygenerowanego drzewa*.

4.2.3. Diagram sekwencji rozgrywki

Rysunek 4.5 ukazuje diagram sekwencji rozgrywki w trybie *Gracz versus PC*.

Na diagramie przedstawiona jest istota komunikacji głównych modułów podczas rozgrywania partii w jedną z dwóch gier. *Aplikacja główna*, *Gra* i *Algorytm*.

Użytkownik końcowy za pomocą menu aplikacji głównej może ustawić parametry gry i uruchomić 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 na odwrót). Po zakończeniu rozgrywki gra zwraca swój stan, który jest możliwy do zobaczenia przez użytkownika poprzez okno aplikacji głównej. Dodatkowo, po każdym wykonanym ruchu, za pomocą modułu *Wizualizacja* generowane jest drzewo stanów algorytmu UCT, a następnie pokazywane jest ono użytkownikowi.



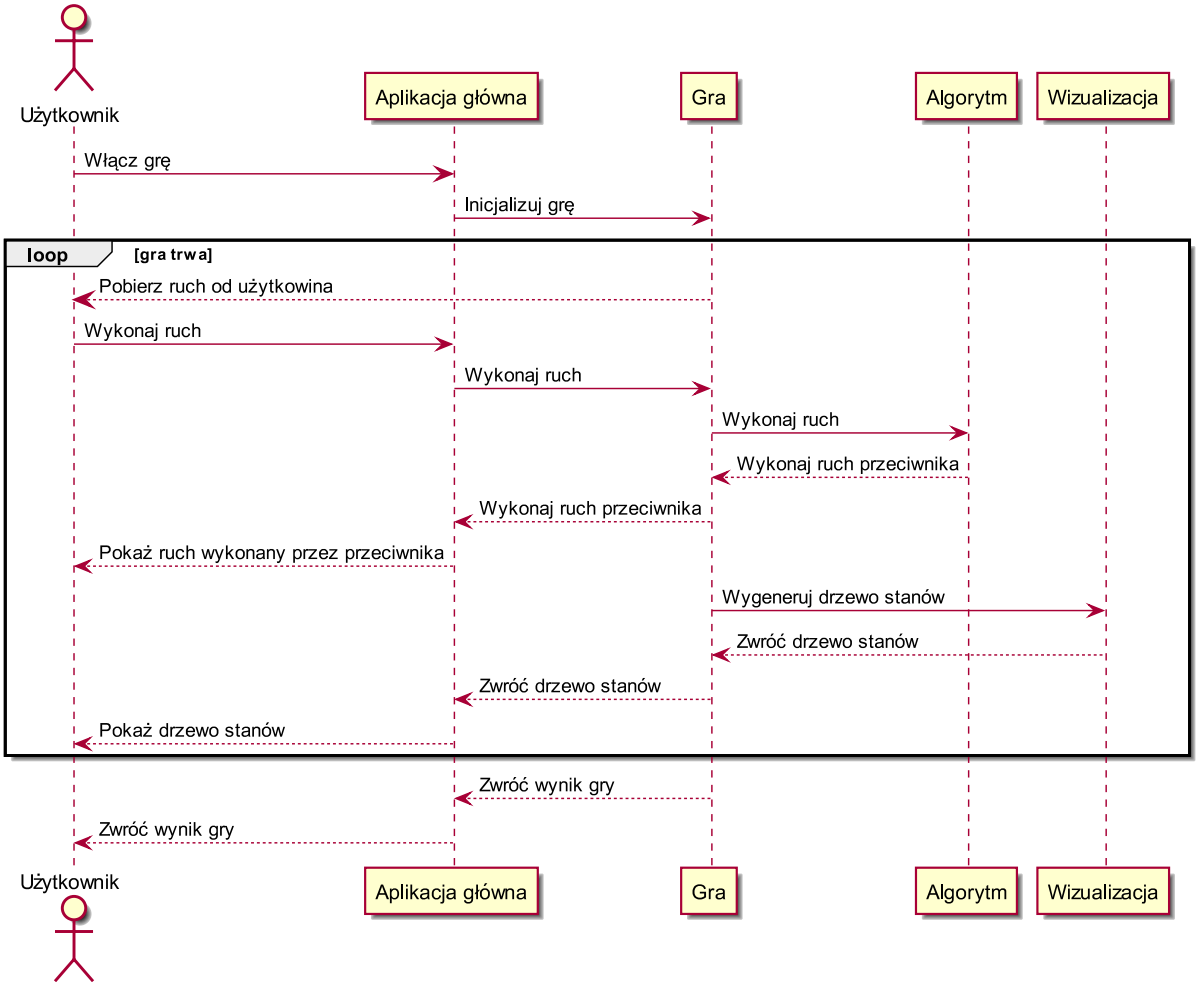
Rysunek 4.4: Diagram stanów aplikacji

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

4.2.4. Diagram sekwencji eksportu drzewa

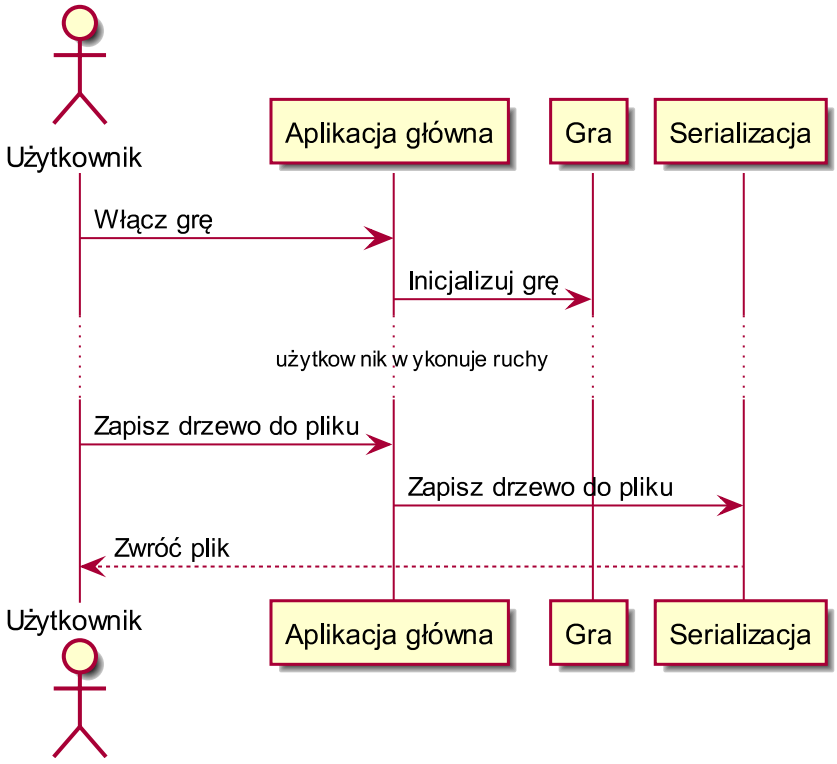
Rysunek 4.6 przedstawia proces współpracy różnych komponentów aplikacji w celu wyeksportowania wygenerowanego przez algorytm drzewa podczas rozgrywki. Proces uruchamiania gry i wykonywania ruchów jest analogiczny do tego na rysunku 4.5.

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 wybranego przez użytkownika folderu.



Rysunek 4.5: Diagram sekwencji rozgrywki

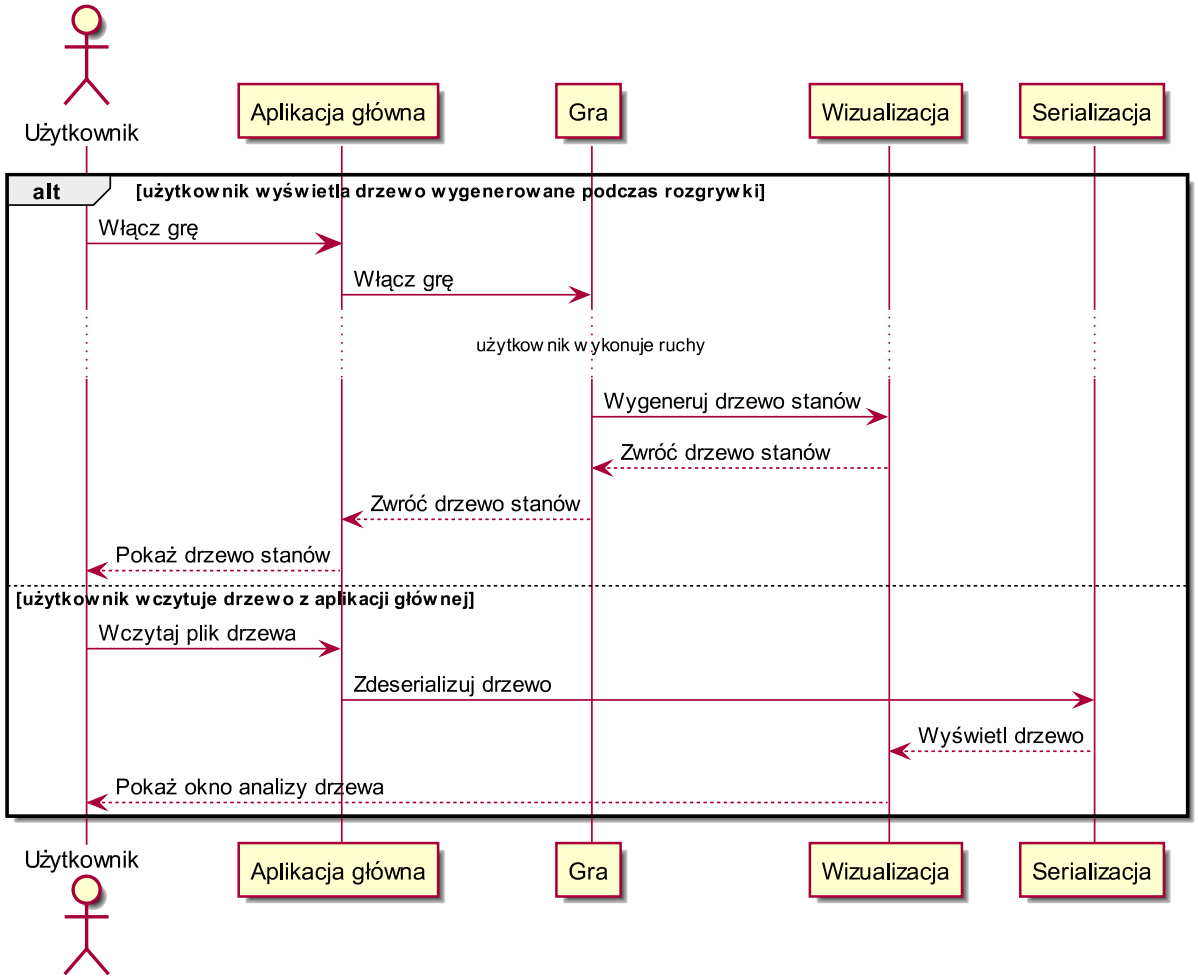
Jest to diagram dla trybu rozgrywki *Gracz versus PC*, jednak w przypadku ustawienia *PC versus PC* sposób serializacji się nie zmienia i diagram wygląda identycznie.



Rysunek 4.6: Diagram sekwencji eksportu drzewa

4.2.5. Diagram sekwencji wizualizacji drzewa

Rysunek 4.7 przedstawia proces wyświetlania wizualizacji drzewa przez użytkownika jako współpracę poszczególnych komponentów aplikacji. Ponownie, proces uruchamiania gry i wykonywania ruchów wygląda tak jak na rysunku 4.5.



Rysunek 4.7: Diagram sekwencji wizualizacji drzewa

Istotny jest fakt, że użytkownik może oglądać drzewa stanów wybierając pliki drzew z dysku w menu głównym aplikacji, ale też oglądać je bezpośrednio podczas rozgrywki po wykonanym ruchu algorytmu. Gdy żądanie jest z poziomu rozgrywki, komponent *Aplikacja główna* komunikuje się z komponentem *Wizualizacja*, który generuje aktualne drzewo i pokazuje je użytkownikowi na ekranie.

Drugi sposób, czyli wczytanie plików drzew z menu głównego, wymaga wcześniejszego zdeserializowania ich 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 wygenerowanymi drzewami.

5. Interfejs użytkownika

5.1. Opis dostępnych okien

5.1.1. Wstęp

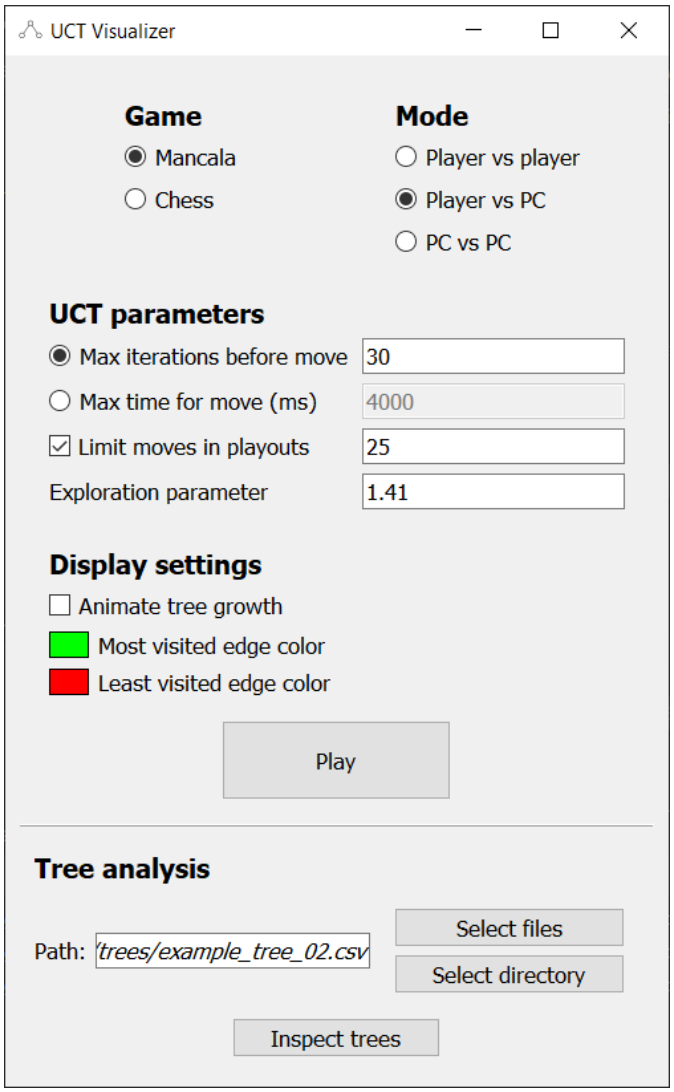
Graficzny interfejs użytkownika składa się z trzech głównych okien, a logika jego działania jest w całości zawarta w module *Aplikacja główna*. Zadaniem graficznego interfejsu jest umożliwienie uruchomienia poszczególnych modułów użytkownikom końcowym.

5.1.2. Menu główne

Ukazane na rysunku 5.1 menu główne jest głównym oknem aplikacji i jest to pierwsze okno ukazywane użytkownikowi po uruchomieniu programu.

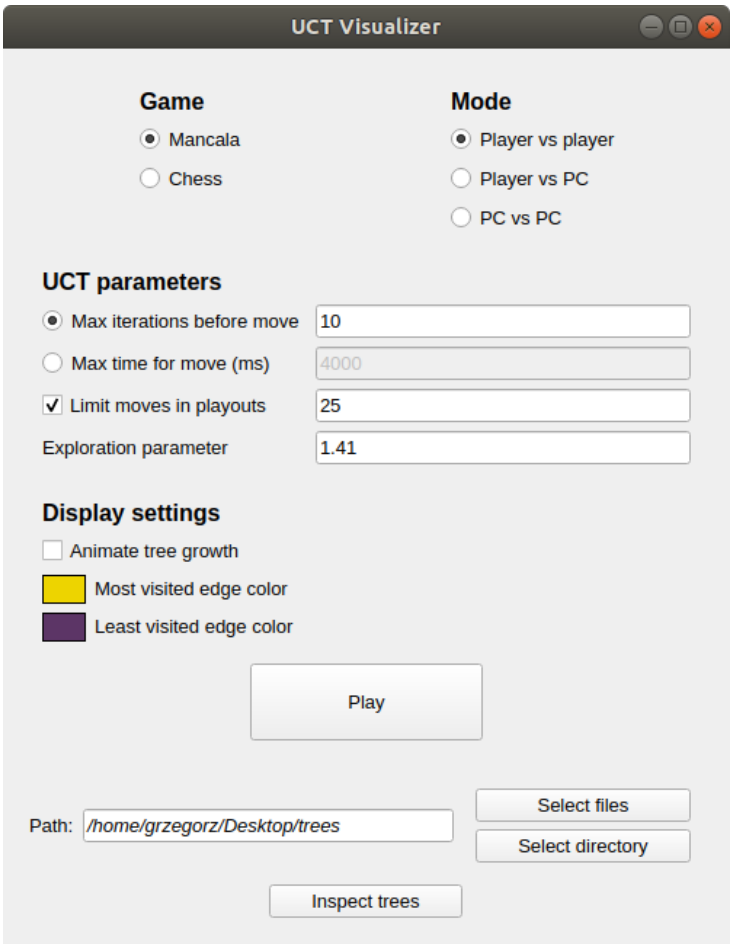
Z poziomu *Menu głównego* użytkownik może przejść do okien *Rozgrywka* i *Analiza drzewa*. W celu rozegrania gry, należy nacisnąć przycisk *Play*. Powyżej przycisku *Play* znajdują się opcje, które pozwolą użytkownikowi ustawić parametry gry dostosowane do jego preferencji.

- **Wybór gry** – do dyspozycji gracza oddane są dwie gry – szachy i mankala.
- **Wybór trybu rozgrywki** – użytkownik może wybrać jeden z trzech trybów gry, opisanych poniżej.
  - **Gracz versus PC** – w tym trybie gracz ma okazję zmierzyć się z komputerem, oglądając przy tym generowane drzewa stanów algorytmu UCT.
  - **PC versus PC** – rozgrywka komputera z samym sobą, z możliwością analizowania wynikowych drzew stanów algorytmu. Wybór tego trybu oznacza brak możliwości wykonywania samodzielnych ruchów przez użytkownika, jednakże to od niego będzie zależało, kiedy algorytm wykona kolejny ruch – będzie miał do dyspozycji przycisk, za pomocą którego będzie mógł powodować postęp w rozgrywce.
  - **Gracz versus gracz** – tryb bez udziału algorytmu UCT, a co za tym idzie, również bez wizualizacji.



Rysunek 5.1: Okno menu głównego - Windows

- **Wybór liczby iteracji algorytmu** – liczba powtórzeń wykonania przez algorytm czterofazowej iteracji metody MCTS opisanej w podrozdziale **??**.
- **Wybór maksymalnego czasu** – podany w milisekundach; czas, po którym komputer będzie przerywał obliczenia i wykona ruch.
- **Ustawienie limitu ruchów** – opcjonalne pole, w którym użytkownik może ustalić maksymalną liczbą ruchów w fazie symulacji algorytmu UCT, w celu wcześniejszego kończenia niepożądanie długich rozgrywek.
- **Ustawienie wartości parametru eksploracji** – pole, w którym użytkownik może ustalić wartość parametru eksploracji algorytmu UCT. Wpływ parametru na działanie algorytmu został opisany w rozdziale 3.2.1.
- **Animowanie rozrostu drzewa** – wybór tej opcji sprawia, że drzewo stanów algorytmu



Rysunek 5.2: Okno menu głównego - Linux

zostaje wyświetlone nie tylko po każdym wykonanym ruchu przez komputer, ale też po każdej iteracji. Co więcej, w panelu po prawej stronie dynamicznie aktualizować się też będzie wartość liczby wszystkich wierzchołków drzewa.

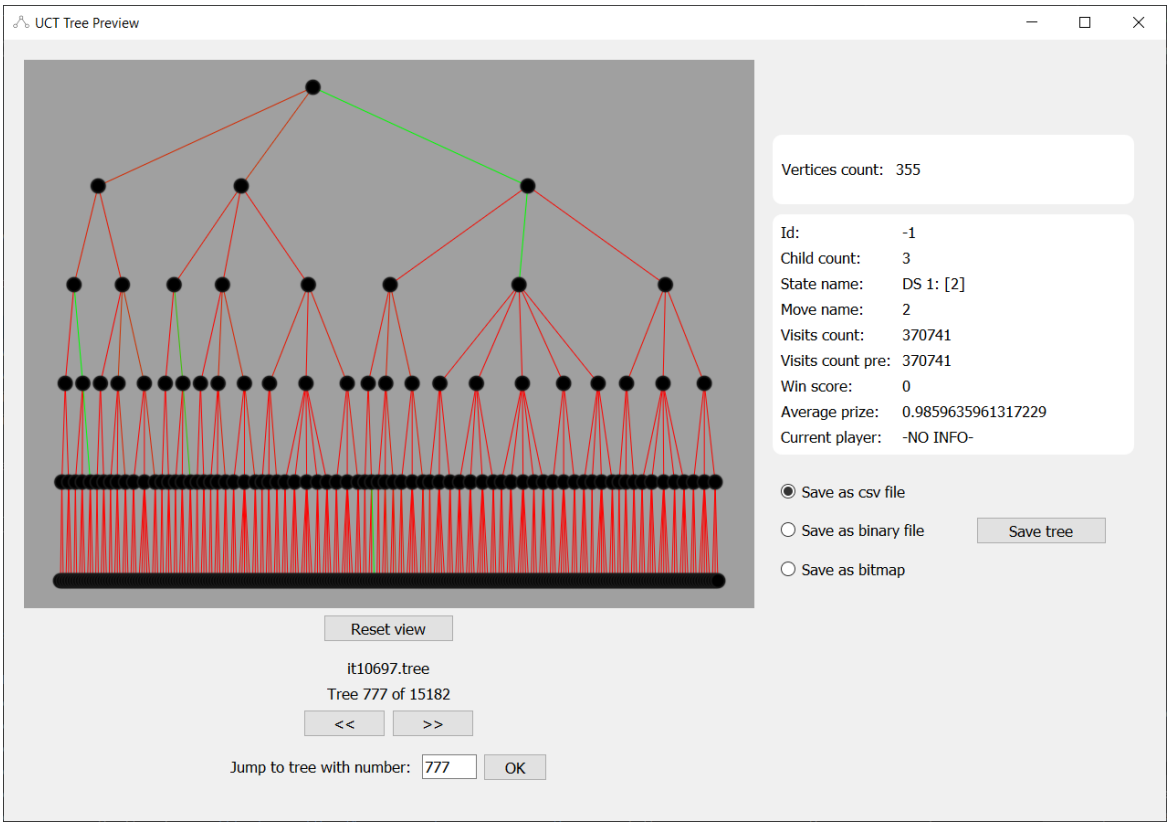
- **Wybór kolorów krawędzi drzewa** - po kliknięciu na któryś z widocznych kolorów ukaże się paleta systemowa, z której można wybrać inne kolory krawędzi. Najczęściej i najrzadziej odwiedzane węzły drzewa przyjmą wskazane przez użytkownika kolory, inne zaś — kolory z ich gradientu, proporcjonalnie do wartości skrajnych licznika odwiedzin.

Drugą kluczową opcją dostępną z menu głównego aplikacji jest możliwość przejścia do okna analizy drzewa lub sekwencji drzew. Jest ona dostępna po naciśnięciu przycisku *Inspect trees*. Użytkownik może wczytać pliki z dysku na dwa sposoby: wybierając pliki z folderu bezpośrednio lub wybierając cały katalog, w którym się one znajdują. Do tego celu służą osobno dwa przyciski: *Select files* i *Select directory*, które znajdują się na prawo od pola tekstowego, w którym wyświetlana jest pełna ścieżka do aktualnie wybranego pliku lub katalogu. W przypadku wybrania większej liczby plików, pokazywana jest ich liczność. Można jednocześnie wybierać

pliki w formacie CSV i binarnym (.tree), których schemat serializacji został opisany w rozdziale 4.1.5. W przypadku wybrania nieprawidłowego katalogu lub niepoprawnych plików, na ekranie użytkownika pojawi się błąd informujący o problemie, który nastąpił.

5.1.3. Analiza drzewa

W oknie ukazanym na rysunku 5.3 będziemy mogli oglądać i analizować drzewa wczytane z plików. Ekran podglądu drzewa pozwala w wygodny sposób analizować i porównywać wczytane drzewa oraz udostępnia opcje wymienione poniżej.



Rysunek 5.3: Okno analizy drzewa

1. **Wyświetlenie statystyk węzła** – po naciśnięciu na węzeł lewym przyciskiem myszy, panel z prawej strony zostaje uzupełniony informacjami, które przechowuje węzeł. Dokładność wykrywania wybranego węzła nie jest zaburzona podczas operowania na przybliżonym lub oddalonym drzewie.

2. **Przybliżanie i oddalanie widoku** – należy naprowadzić kursor na szary obszar drzewa, a następnie przewijać środkowym przyciskiem myszy w górę lub w dół.

3. **Przesuwanie drzewa** – gdy kursor znajduje się w obszarze drzewa, należy nacisnąć prawy

przycisk myszy, przytrzymać i poruszać nim na boki w celu przesunięcia widoku całego drzewa.

4. **Reset pozycji drzewa** – pierwotną pozycję i rozmiar drzewa można przywrócić za pomocą przycisku *Reset view* znajdującego się pod widokiem drzewa.

5. **Zmiana drzewa w sekwencji** – jeżeli wczytane zostało więcej niż jedno drzewo, kluczową funkcją jest opcja wygodnego przełączania się pomiędzy nimi. Można to zrobić na trzy sposoby: używając przycisków $<<$ i $>>$, które znajdują się w dolnej części okna; klikając lewy i prawy przycisk myszy na klawiaturze (czasami trzeba ustawić uprzednio uaktywnić obszar drzewa, klikając na niego) oraz użycie pola tekstowego, pozwalającego na zmianę aktualnie wyświetlanego drzewa i kliknięcie przycisku *OK*. W przypadku wyświetlania pierwszego lub ostatniego drzewa w sekwencji odpowiednie przyciski strzałek stają się nieaktywne, a w przypadku pola tekstowego ze skokiem następuje walidacja, czy podany numer drzewa mieści się w zakresie.

6. **Zapis drzewa** - możliwość zapisu wyświetlanego drzewa w jednym z trzech dostępnych formatów: CSV, binarnym lub rastrowym (.png). Opcja ta znajduje się z prawej strony okna, pod panelem zawierającym wartości przechowywane przez węzeł. Podczas eksportu do grafiki rastrowej zadbano o to, aby nie występowały artefakty związane z dużym zagęszczeniem krawędzi. Taka grafika zachowuje też aktualne powiększenie i przesunięcie drzewa – działa na zasadzie zrzutu ekranu.

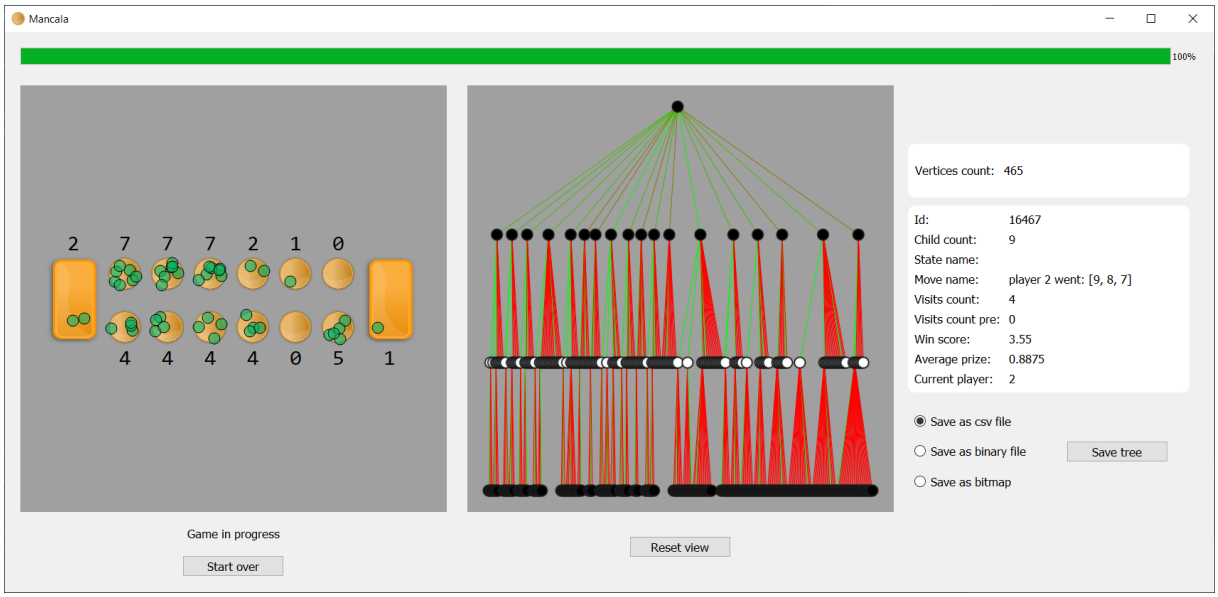
Co więcej, dla każdego drzewa w sekwencji w panelu dolnym wyświetla się informacja o tym, którym z kolei na liście jest oglądane drzewo oraz jaka jest nazwa pliku, z którego zostało ono wczytane. Dodatkowo, wyświetlana jest liczba wszystkich wczytanych drzew.

5.1.4. Rozgrywka

Na rysunku 5.4 przedstawiony jest interfejs graficzny podczas rozgrywki. Środkową i prawą część okna zajmuje komponent dotyczący analizy drzewa, który został opisany w rozdziale 5.1.3. W związku z tym, wszystkie wyżej opisane funkcjonalności mają zastosowanie również w oknie rozgrywki, poza aspektem dotyczącym sekwencji drzew. Z lewej strony znajduje się panel z wybraną wcześniej grą. Funkcjonalności okna *Rozgrywka* zostały opisane poniżej.

1. **Wykonywanie ruchów** przez gracza, kiedy gra przeciwko PC:

   (a) w przypadku szachów - wybierając odpowiednią figurę na szachownicy, a następnie jeden z wyświetlonych dostępnych ruchów. Gracz w tym trybie zawsze gra białymi,

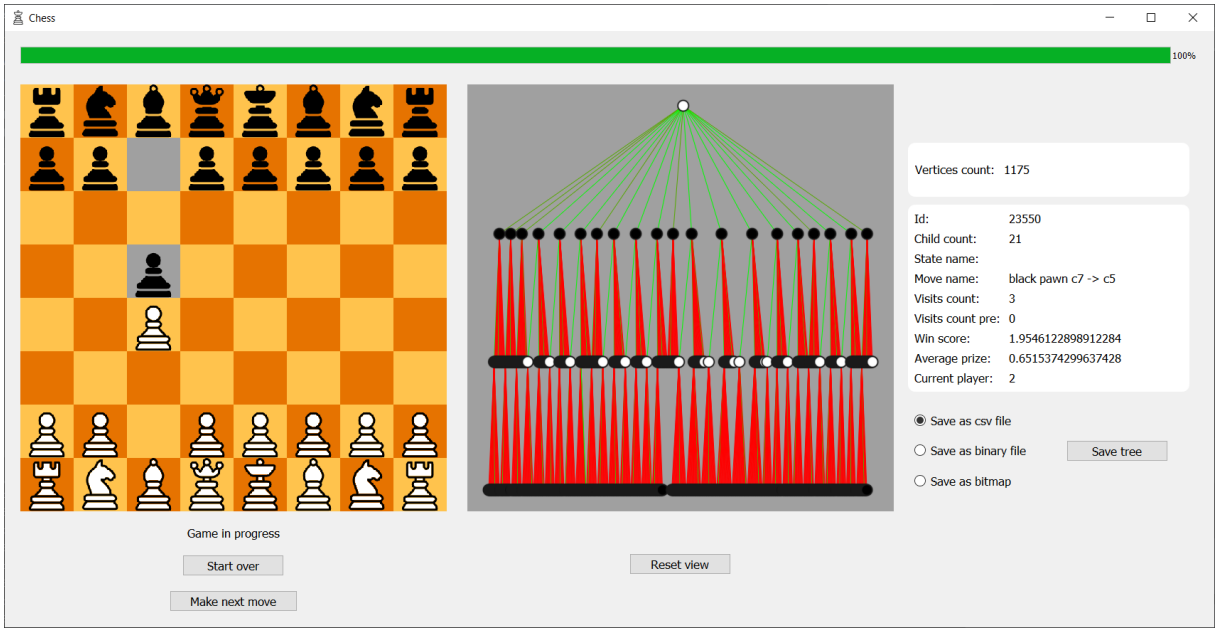


Rysunek 5.4: Okno rozgrywki – mankala, gracz przeciwko PC

(b) w przypadku mankali - wybierając pewien niepusty dołek z dolnego rzędu planszy.

2. **Zrestartowanie gry** za pomocą przycisku *Start over* w lewej dolnej części okna,

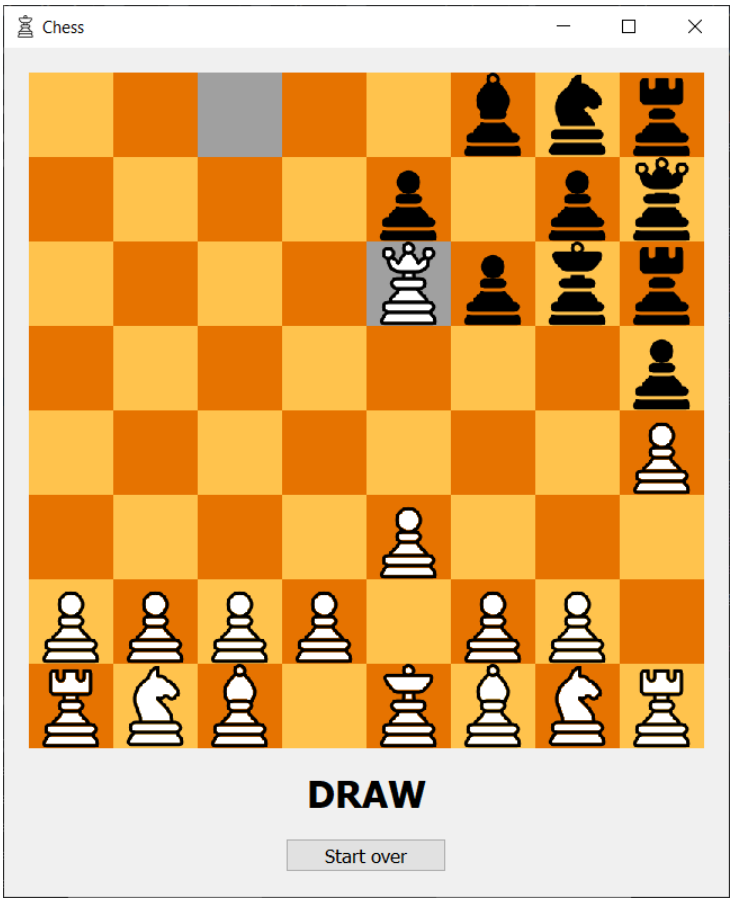
3. **Zapisanie drzewa do pliku** za pomocą przycisku *Save tree*. Format zapisywanego pliku wybierany jest poprzez zaznaczenie odpowiedniej opcji w liście znajdującej się na lewo od przycisku *Save tree*.



Rysunek 5.5: Okno rozgrywki – szachy, PC przeciwko PC

Bezpośrednio pod obszarem gry znajduje się etykieta z napisem dotyczącym obecnego stanu

rozgrywki. *Game in progress* oznacza, że gra się toczy i wciąż można wykonywać ruchy. W przypadku zakończenia rozgrywki, obszar gry lub przycisk *Make next move* zostaje dezaktywowany i nie można wykonywać kolejnych ruchów aż do ponowienia gry, a etykieta będzie informować o tym, który gracz zwyciężył, wyświetlając napis *Player 1 wins*, *Player 2 wins* lub *Draw*. Przykładowy stan zakończonej gry pokazany jest na rysunku 5.6.



Rysunek 5.6: Okno rozgrywki – szachy, dwóch graczy

6. Instrukcje

6.1. Instalacja

6.1.1. Wymagania sprzętowe

W celu wykorzystania możliwości, które daje prezentowany system, należy uruchomić go na komputerze, który spełnia wymienione poniżej wymagania sprzętowe.

1. Procesor: Intel Core i5-3470 3.2 GHz / AMD FX-8350 4 GHz.
2. Pamięć RAM: 8 GB.
3. Karta graficzna: Nvidia GTX 660 2GB / AMD HD 7870 2 GB.
4. Miejsce na dysku twardym: 150 MB.
5. System operacyjny: Windows 10 / Ubuntu 16.04.

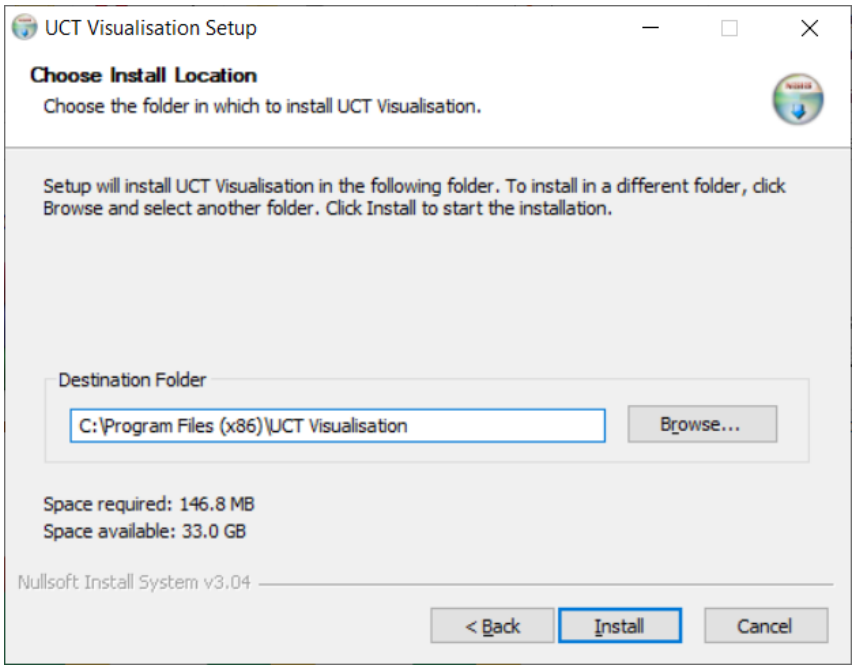
6.1.2. Instrukcja instalacji Windows

Aby zainstalować aplikację na dysku lokalnym, należy uruchomić plik instalacyjny *UCTVisualisationSetup.exe*. Następnie ukaże się okno przeprowadzające użytkownika przez proces instalacji. Należy wybrać folder docelowy i kliknąć przycisk *Install*. Po zakończeniu procesu można zamknąć okno instalacji, zaznaczając przedtem *Run UCT Visualisation*, aby bezpośrednio po zamknięciu instalatora uruchomiła się aplikacja.

Zainstalowana aplikacja znajduje się w folderze wskazanym przez użytkownika podczas instalacji w katalogu *UCT Visualisation*. Aby uruchomić aplikację należy w niego wejść i uruchomić plik *UCT Visualisation.exe*. Okno instalacji powinno przypominać to, ukazane na rysunku 6.1.

6.1.3. Instrukcja instalacji Linux

W celu instalacji programu na lokalnym dysku, potrzebne jest uruchomienie pliku *UCTVisualisationSetup.deb*, a następnie kliknięcie przycisku *Install*, tak jak na rysunku 6.2. Aplikacja



Rysunek 6.1: Okno instalacji programu - Windows

o nazwie *UCT Visualisation* znajduje się w katalogu o takiej samej nazwie i domyślnie zostaje zainstalowana pod ścieżką:

```
/opt/"UCT Visualization"
```

zatem w celu uruchomienia aplikacji należy wpisać w systemowym terminalu następującą komendę:

```
/opt/"UCT Visualization"/"UCT Visualization"
```

lub:

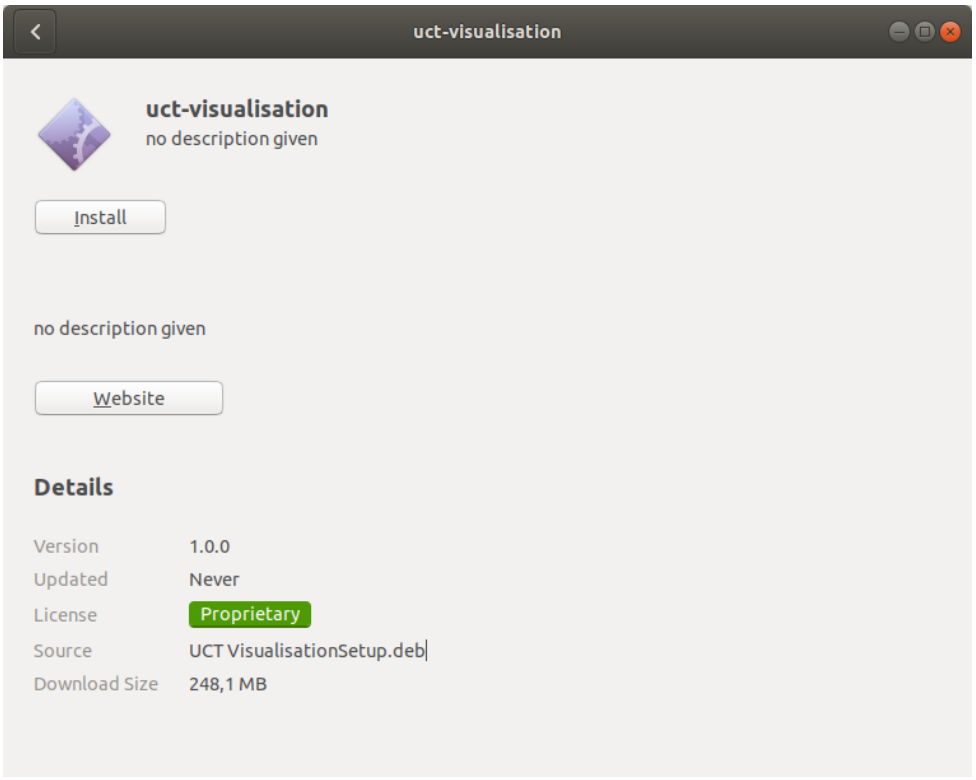
```
cd /opt/"UCT Visualization"
./"UCT Visualization"
```

6.2. Mankala

6.2.1. Wstęp

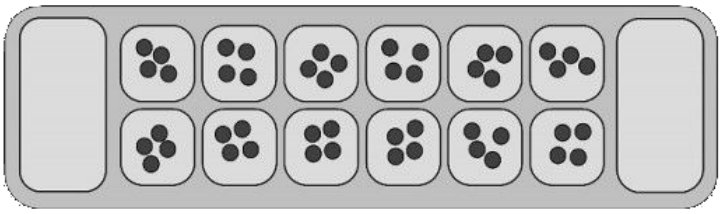
Mankala, jako jedna z przykładowych gier modułu *Gry*, jest mniej popularną grą logiczną, więc postanowiono przedstawić instrukcję grania w nią.

Plansza mankali zawiera dwanaście mniejszych pól i dwa większe, nazywane domami. Każdy z graczy ma sześć mniejszych pól leżących przed nim i dom po jego prawej stronie. Na początku gry w każdym z pól znajdują się cztery kamienie. Celem obu graczy jest zebranie w domu jak



Rysunek 6.2: Okno instalacji programu - Ubuntu

największej liczby kamieni w ich domach. Gra kończy się, gdy wszystkie pola jednego z graczy są puste. Wtedy pozostałe kamienie przydzielane są drugiemu graczowi.



Rysunek 6.3: Plansza mankali

6.2.2. Ruch gracza

Ruch gracza polega na wyjęciu wszystkich kamieni z wybranego własnego pola i rozdysponowaniu po jednym do kolejnych pól, omijając dom przeciwnika. Rozdysponowanie jest wykonywane w kierunku odwrotnym do ruchu wskazówek zegara. Ponadto, jeśli ostatni kamień wyląduje we własnym domu – gracz musi wykonać kolejny ruch. Jeśli nie – następuje ruch przeciwnika.

6.2.3. Bicie

Ostatnią zasadą mankali jest bicie. Bicie następuje, jeśli po wykonaniu ruchu, ostatni kamień

wyląduje na pustym polu gracza. W takiej sytuacji gracz zabiera wszystkie kamienie z przyległego pola po drugiej stronie planszy. Jeżeli w sąsiednim polu nie ma kamieni, żaden kamień nie zostaje zbijany. Tak samo, ruch nie kończy się biciem, jeżeli ostatni kamień wyląduje na pustym polu przeciwnika.

7. Podsumowanie i ocena

7.1. Weryfikacja rezultatów

7.1.1. Uzyskane efekty

Dostarczany produkt jest sprawnie działającym i wygodnym narzędziem do wizualizowania i analizowania drzew stanów algorytmu UCT. Pozwala on również na ich generowanie podczas rozgrywki przeciwko komputerowi w jedną z dwóch zaimplementowanych gier logicznych. Dodatkowo, wszystkie opisywanie wcześniej funkcjonalności spaja intuicyjny i uporządkowany interfejs graficzny.

7.1.2. Testy akceptacyjne

Testy akceptacyjne zostały przeprowadzone w celu sprawdzenia, czy aplikacja spełnia założenia opisane w dokumentacji wymagań projektu. Test 7.1 konfrontuje założenia modułu *Gry*, test 7.2 – modułu *Serializacja*, a pozostałe testy weryfikują założenia modułu *Wizualizacja*.

Testy akceptacyjne zostały wykonane na komputerze:

- z zainstalowanym systemem operacyjnym *Windows 10 Education N*,
- z zainstalowanym interpreterem języka *Python 3.7.2* i biblioteką *PyQt5*,
- wyposażonym w procesor *Intel Core i7-8700k @3.70 GHz*,
- wyposażonym w kartę graficzną *NVIDIA GeForce GTX 1060 6GB*,
- wyposażonym w 32GB pamięci RAM.

Pierwszy z przeprowadzonych testów dotyczy funkcjonalności modułu *Gry*, a wynik opisany został w tabeli 7.1.

Tablica 7.1: Raport z pierwszego testu

Testowane wymaganie	*Użytkownik będzie mógł wybrać jedną z dwóch przykładowych gier, a do wyboru będzie miał trzy tryby rozgrywki.*
Kroki testowe	1. Z menu głównego aplikacji wybierz opcję *Chess*. 2. Z menu głównego aplikacji wybierz opcję *Player vs player* i sprawdź tryb rozgrywki dla dwóch graczy. 3. Z menu głównego aplikacji wybierz opcję *Player vs PC* dla różnych ustawień algorytmu UCT. 4. Z menu głównego aplikacji wybierz opcję *PC vs PC* dla różnych ustawień algorytmu UCT. 5. Z menu głównego aplikacji wybierz *Mancala* i powtórz kroki 2–5.
Wynik	Pozytywny.

Drugi z przeprowadzonych testów dotyczy funkcjonalności modułu *Serializacja*, a wynik opisany został w tabeli 7.2.

Tablica 7.2: Raport z drugiego testu

Testowane wymaganie	*Użytkownik będzie mógł zapisać analizowane drzewa do pliku csv, do pliku binarnego oraz do bitmapy.*
Kroki testowe	1. Z menu głównego aplikacji wybierz ścieżkę do dowolnego pliku z zserializowanym drzewem. 2. Naciśnij przycisk *Inspect tree*. 3. Naciśnij przycisk *Save to csv file*. 4. Naciśnij przycisk *Save to binary file*. 5. Naciśnij przycisk *Save to bitmap file*. 6. Sprawdź, czy bitmapa wygenerowana w kroku 5 odpowiada drzewu z pliku początkowego. 7. Z menu głównego aplikacji wybierz ścieżkę plików wygenerowanych w kroku 3 i 4, żeby sprawdzić, czy zapisane drzewa wizualizowane są tak samo jak w początkowym pliku.
Wynik	Pozytywny.

Trzeci z przeprowadzonych testów dotyczy funkcjonalności modułu *Wizualizacja*, a wynik opisany został w tabeli 7.3.

Tablica 7.3: Raport z trzeciego testu

Testowane wymaganie	*Użytkownik będzie mógł wyświetlić informacje związane z wybranym węzłem drzewa, a także przybliżać i oddalać cały graf.*
Kroki testowe	1. Z menu głównego aplikacji wybierz ścieżkę do dowolnego pliku z drzewem. 2. Naciśnij przycisk *Inspect tree*. 3. Przy użyciu prawego przycisku myszki chwyć za obszar rysowania i poruszaj się po wizualizacji. 4. Używając kółka myszki, przybliż i oddal wizualizowane drzewo. 5. Kliknij dowolny wierzchołek drzewa lewym przyciskiem myszki i sprawdź, czy panel z prawej strony wyświetla informacje związane z wybranym wierzchołkiem.
Wynik	Pozytywny.

Czwarty z przeprowadzonych testów dotyczy wydajności modułu *Wizualizacja*, a wynik opisany został w tabeli 7.4.

Tablica 7.4: Raport z czwartego testu

Testowane wymaganie	*Dla drzew do 100 000 wierzchołków wizualizacja nie powinna zajmować więcej niż 3s.*
Kroki testowe	1. Z menu głównego aplikacji wybierz ścieżkę do pliku *tree_100k.csv*. 2. Naciśnij przycisk *Inspect tree*.
Wynik	Pozytywny – deserializacja, ulepszony algorytm Walkera i wyświetlenie drzewa z pliku zajęło 2.802s.

Piąty z przeprowadzonych testów dotyczy wydajności modułu *Wizualizacja*, a wynik opisany został w tabeli 7.5.

Tablica 7.5: Raport z piątego testu

Testowane wymaganie	*Dla drzew do 250 000 wierzchołków wizualizacja nie powinna zajmować więcej niż 5s.*
Kroki testowe	1. Z menu głównego aplikacji wybierz ścieżkę do pliku *tree_250k.csv*. 2. Naciśnij przycisk *Inspect tree*.
Wynik	Pozytywny – deserializacja, ulepszony algorytm Walkera i wyświetlenie drzewa z pliku zajęło 4.626s.

Wszystkie testy akceptacyjne zakończyły się pozytywnie, a więc wymagania zostały spełnione.

7.2. Dalszy rozwój

7.2.1. Kontynuacja pracy

Aplikacja spełnia wszystkie początkowe wymagania określone w rozdziale 2.2, jednak w przyszłości mogą zostać do niej dodane usprawnienia. W celu dalszej poprawy jakości i zwiększenia liczby możliwych opcji, obecny produkt można rozszerzyć o następujące funkcjonalności:

1. **Dodanie kolejnych gier** - obecna architektura projektu umożliwia rozszerzenie aplikacji o kolejne gry logiczne, które spełniają założenia algorytmu UCT. Zwiększenie liczby dostępnych gier daje więcej możliwości badania algorytmu UCT.

2. **Usprawnienie i zrównoleglenie symulacji szachów** - symulacja ruchu szachowego przez komputer zajmuje dużo więcej czasu w przypadku szachów niż mankali. Zrównoleglenie pewnych obliczeń szachowych, które zajmują najwięcej czasu, poprawiłoby jakość decyzji podejmowanych przez algorytm w jednostce czasu.

3. **Dodanie otwarć szachowych** - algorytm UCT dla niewielkiej liczby iteracji ma tendencję do wybierania ruchów dających natychmiastowe wynagrodzenie, tym samym ignorując ruchy, które niebezpośrednio prowadzą do lepszych, złożonych zagrań. To oznacza, że algorytm podejmuje lepsze decyzje w końcowym stadium rozgrywki, niż w początkowym —

szczególnie dobrze widać to w przypadku rozgrywki szachowej. Możliwe byłoby zatem wprowadzenie usprawnienia w postaci zbioru sprawdzonych szachowych rozwiązań dotyczących rozpoczęcia rozgrywki, z których korzystałby komputer przy wyznaczaniu początkowych ruchów.

4. **Lepsza ewaluacja wartości figur w szachach** - sposób wartościowania figur jest miejscem, w którym można znacznie rozwinąć potencjał decyzji podejmowanych przez algorytm UCT. Na ten moment każda figura ma arbitralnie ustaloną wartość, bez względu na swoje położenie na szachownicy. Tymczasem, wartość danej figury może zależeć od wielu czynników, między innymi od pozycji, w której się znajduje. Dla przykładu, koń w centrum planszy będzie miał większy potencjał od takiego, który umieszczony jest w rogu. Tak samo pion, który jest bliżej ostatniego rzędu pól, jest bardziej wartościowy od piona na swojej początkowej pozycji, a pion zdublowany prawdopodobnie jeszcze mniej. Liczba konfiguracji i czynników wpływających na potencjał figury w danym miejscu na szachownicy wykracza poza temat tej pracy, jednak potencjalne rozwinięcie lepiej przemyślanej ewaluacji figur wpłynęło by korzystnie na zdolność algorytmu do podejmowania bardziej obiecujących ruchów.

5. **Inteligentne przydzielanie pamięci dla drzew w sekwencji** - podczas przełączania się pomiędzy kolejnymi drzewami w sekwencji, za każdym razem następuje wczytanie do pamięci podręcznej wartości i pozycji wszystkich węzłów danego drzewa. Oznacza to, że jeśli użytkownik analizuje dwa kolejne drzewa i przełącza między nimi na zmianę, za każdym razem te same drzewa są wczytywane do pamięci podręcznej na nowo. Usprawnieniem tego procesu byłoby przechowywanie wcześniej już wczytanych drzew, jednocześnie automatycznie kontrolując ilość wykorzystanych zasobów. Co więcej, program mógłby wczytywać od razu pewną liczbę kolejnych drzew z wyprzedzeniem. Takie operacje zaoszczędziłyby czas użytkownika, zwłaszcza podczas wczytywania drzew z dużą ilością węzłów.

Czynnikiem wpływającym negatywnie na szybkość wykonywania obliczeń w języku Python jest fakt, że jest to język interpretowany. W celu przyspieszenia modułu odpowiedzialnego za grę w szachy, można by przepisać go w całości z języka Python do języka kompilowanego.

8. Wnioski

Praca nad projektem powiązanym z tematem sztucznej inteligencji pozwoliła zrozumieć nam dokładną zasadę działania algorytmu UCT i sprawiła, że zgłębiliśmy naszą wiedzę na temat jednego z popularniejszych algorytmów dotyczących symulacji obiecujących ruchów komputera w grach logicznych.

Proces implementacji wizualizacji drzewa stanów uświadomił nam, jak bardzo wydajne może być zastosowanie karty graficznej, w celu wykonywania sprawnych obliczeń związanych z grafiką. Utwierdził nas on też w przekonaniu, jak dużo możliwości daje korzystanie z biblioteki *OpenGL*.

Dodatkowo, w naszej pracy nauczyliśmy się efektywnego łączenia komponentów z różnych bibliotek w celu stworzenia jednej spójnej aplikacji. Nauczyło nas to także sprawnego korzystania z zewnętrznych dokumentacji. Zauważyliśmy również, że biblioteki przyciągające programistów swoją łatwością w obsłudze nie zawsze są najlepszym wyborem. Za przykład może posłużyć używana przez nas biblioteka *PyGame* do graficznej reprezentacji gier. Ostatecznie zrezygnowaliśmy z niej, ponieważ nie była ona kompatybilna z innymi komponentami, między innymi z modułem *PyQt*, odpowiedzialnym za tworzenie interfejsu graficznego.

Podczas pracy zrozumieliśmy, że trendy panujące na rynku informatycznym dotyczące języków programowania nie zawsze współgrają z ich wydajnością. Przykładowo, Python ze względu na bycie językiem interpretowanym nie jest najlepszym wyborem do programów wymagających wykonywania bardzo dużej ilości obliczeń i ustępuje szybkością starszym językom programowania, tak jak na przykład C. Jest on natomiast dobrym wyborem do zadań, które wymagają napisania kodu szybkiego, lecz niekoniecznie optymalnie działającego.

Zmierzyliśmy się również z problemem dotyczącym rysowania drzew nieprzecinających się krawędziami, w sposób możliwie zwięzły. Okazało się, że znaleziony przez nas w opracowaniach naukowych ulepszony algorytm Walkera nie wydaje się bardzo popularny, mimo swojego czasu obliczeń rzędu $\Theta(n)$. Uświadomiło nas to, jak wiele wydajnych, a jednocześnie mało znanych

rozwiązań można znaleźć w opracowaniach naukowych.

Bibliografia

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

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

[3] K. Marriott, *NP-Completeness of Minimal Width Unordered Tree Layout*, Journal of Graph Algorithms and Applications, vol. 8, no. 3, pp. 295-312 (2004).

Spis rysunków

Spis tabel

Spis załączników

1. Załącznik 1
2. Załącznik 2
3. Jak nie występują, usunąć rozdział.