UNO MCTS

**Wersja 1.1**

Autor: Rafał Topolski – 113464 Informatyka II rok stacjonarne

Spis treści

[Przeznaczenie dokumentu 2](#_Toc75110208)

[Ogólne założenia projektu 2](#_Toc75110209)

[Opis i zasady gry Uno 2](#_Toc75110210)

[Wstęp teoretyczny 3](#_Toc75110211)

[Schemat blokowy algorytmu 6](#_Toc75110212)

[Opis programu 7](#_Toc75110213)

[Kod algorytmu 13](#_Toc75110214)

[Instalacja programu 26](#_Toc75110215)

[Zawartość folderu zip 26](#_Toc75110216)

[Instrukcja obsługi programu 26](#_Toc75110217)

[Bibliografia 28](#_Toc75110218)

[Oświadczenia 29](#_Toc75110219)

# Przeznaczenie dokumentu

Niniejsza dokumentacja ma na celu przedstawienie projektu komputerowej reprezentacji gry karcianej UNO

# Ogólne założenia projektu

Projekt ma na celu stworzenie komputerowej reprezentacji gry karcianej UNO. Gra jest przeznaczona dla 1-4 graczy. Gra pozwoli na dodanie sztucznej inteligencji w rozgrywce. Wykorzystany został algorytm przeszukiwania drzewa Monte Carlo. Gra zaoferuje graficzny okienkowy interfejs i interaktywne elementy. Program umożliwi zapisanie wyników rozgrywki.

# Opis i zasady gry Uno

Uno to gra karciana stworzona w roku 1971 przez Merle’a Robbinsa. Talia do gry składa się z 108 kart i składa się z:

**Kart zwykłych:**

19 – kart czerwonych ponumerowanych od 0 do 9 (jedna karta z numerem 0 oraz po dwie karty z numerami od 1 do 9)

19 – kart niebieskich ponumerowanych od 0 do 9 (jedna karta z numerem 0 oraz po dwie karty z numerami od 1 do 9)

19 – kart zielonych ponumerowanych od 0 do 9 (jedna karta z numerem 0 oraz po dwie karty z numerami od 1 do 9)

19 – kart żółtych ponumerowanych od 0 do 9 (jedna karta z numerem 0 oraz po dwie karty z numerami od 1 do 9)

**Kart akcji:**

8 kart stopu (po dwie na każdy kolor)

8 kart zmiany kierunku (po dwie na każdy kolor)

8 kart weź dwie (po dwie na każdy kolor)

4 czarne karty wybierz kolor

4 czarne karty wybierz kolor + weź cztery

**Zasady gry:**

Rozdaje się każdemu graczowi 7 kart i jedną z talii kładzie się na środek. Gracz musi swoją kartę kolorem, numerem lub symbolem do odkrytej karty. Jeżeli gracz nie posiada żadnej karty pasującej do tej odkrytej musi pociągnąć kartę z talii. Jeśli wyciągnięta karta pasuje do odkrytej, jeszcze w tej samej kolejce gracz może ją dołożyć. Jeżeli nie – ruch ma kolejny gracz. Nie ma przymusu w dokładaniu kart.

Karta stopu – następny gracz traci kolejkę.

Zmiana kierunku – karta zmienia kierunek gry.

Weź dwie – następny gracz dobiera dwie karty z talii (chyba następny gracz posiada kartę „weź dwie” lub „wybierz kolor + weź cztery” lub pierwsza wyciągnięta karta to karta „weź dwie” lub „wybierz kolor + weź cztery”, wtedy kolejkę przekazuje następnemu graczowi z powiększoną liczbą kart do dobierania)

Wybierz kolor – zagrywający gracz deklaruje zmianę koloru na jeden z 4 kolorów kart zwykłych.

Wybierz kolor + weź cztery – zagrywający kartę deklaruje zmianę koloru i następny gracz dobiera 4 karty (chyba następny gracz posiada kartę „weź dwie” lub „wybierz kolor + weź cztery” lub pierwsza wyciągnięta karta to karta „weź dwie” lub „wybierz kolor + weź cztery”, wtedy kolejkę przekazuje następnemu graczowi z powiększoną liczbą kart do dobierania)

Wygrywa gracz który pierwszy pozbędzie się wszystkich kart z ręki.

# Wstęp teoretyczny

Implementacja sztucznej inteligencji wymaga matematycznego zdefiniowania stanu rozgrywki do formy przetwarzalnej przez algorytm. Najbardziej naturalną formą będzie drzewo stanów gry. Gdy algorytm ma do czynienia z grami z pełnym zbiorem informacji, utworzenie drzewa stanów gry jest zadaniem prostym. Taką grą jest kółko i krzyżyk, kiedy wiadomo jaki ruch może wykonać każdy gracz w każdym momencie rozgrywki. Dla takich gier prosty algorytm może sprawdzić każdy możliwy stan i wyznaczyć każdy możliwy wynik rozgrywki. Ten pomysł dokładnie realizuje algorytm przeszukiwania drzewa min-max. Uno, w przeciwieństwie do kółko i krzyżyk, to gra z niepełnym zbiorem informacji, czyli nie wiadomo jakie karty posiadają przeciwnicy. Jest to duże wyzwanie, gdyż algorytm musi rozważyć wszystkie hipotetycznie możliwe stany gry w danym momencie. Stan gry jest dodatkowo bardziej skomplikowany więc symulacja wielu drzew stanów gry, wykracza poza możliwości sprzętowe statystycznego komputera osobistego. Rozwiązaniem tego problemu będzie przeszukiwanie drzewa metodami Monte Carlo. Metody Monte Carlo polegają na losowym próbkowaniu przestrzeni zagadnienia w uzyskania zbieżności z prawdziwą wartością rozwiązania. Oferują one znaczną poprawę wydajnościową względem drzewa min-max. W przypadku przeszukiwania drzewa algorytm przemieszcza się losową ścieżką, aż do osiągnięcia węzłów kończących grę. W skład algorytmu wchodzą 4 kroki powtarzane przez ustaloną ilość razy.

*https://pure.york.ac.uk/portal/files/13014166/CowlingPowleyWhitehouse2012.pdf*

Obraz zawierający zegarek

Opis wygenerowany automatycznie

Rysunek : Schemat algorytmu MCTS [File:MCTS (English) - Updated 2017-11-19.svg - Wikimedia Commons](https://commons.wikimedia.org/wiki/File:MCTS_(English)_-_Updated_2017-11-19.svg)  
oparte na licencji [Creative Commons — Attribution-ShareAlike 4.0 International — CC BY-SA 4.0](https://creativecommons.org/licenses/by-sa/4.0/deed.en)

1. Wybór
2. Rozrost
3. Symulacja
4. Propagacja wsteczna

Krok wyboru jest oparty na podstawie czynnika UCB (upper confidence bound). W skrócie polega to na określeniu wartości kroku i jego niepewności. Wartość to liczba zwcięstw dla decyzji. Niepewność to parametr odwrotnie proporcjonalny do liczby symulacji dla danej decyzji. Niepewność może zaważyć na rzecz tych mniej wartościowych ale i mniej pewnych wyborów. Balans czynnika eksploracji i wykorzystania (exploration and exploitation balance) to ważna część algorytmu. Dzięki wskaźnikowi UCB, algorytm będzie dynamicznie zwiększał częstotliwość symulacji dla obiecujących decyzji i sukcesywnie zmniejszał częstotliwość przeszukiwania tych mniej korzystnych. Wzór matematyczny opisujący wskaźnik przedstawia się następująco:

– Liczba symulacji.

- liczba zwycięstw elementu a.

– liczba symulacji elementu a.

– szacunek wartości dla danego kroku w tym wypadku będzie to stosunek zwycięstw do liczby odwiedzeń elementu.

= Jest to stosunek liczby symulacji elementu a do liczby wszystkich symulacji.

– jest to stała wartość pewności która kontroluje stopień eksploracji decyzji. Im większa wartość tym algorytm bardziej skupia się na symulowaniu tych decyzji które były symulowane mniejszą ilość razy.

Można wyróżnić tam dwie odrębne części:

Wykorzystanie (exploitation) – reprezentowaną przez . UCB opiera się na zasadzie „optymizmu w fakcie niepewności”. Która znaczy to, że jeżeli nie wiemy która decyzja jest najlepsza to podejmujemy tą która wygląda aktualnie najlepiej. Dlatego wyliczana jest wartość dla konkretnej decyzji.

Eksplorację (exploration) – reprezentowana przez drugą część równania . Której stopień jest kontrolowany przez współczynnik c. Ta część równania zapewnia miarę niepewności każdej wyliczonej nagrody. Jeżeli decyzja była symulowana małą ilość razy to będzie bardzo małe, w konsekwencji wyraz niepewności będzie bardzo duży.

*https://www.jmlr.org/papers/volume3/auer02a/auer02a.pdf*

Rozrost to po prostu stworzenie dodatkowych decyzji z wybranego ruchu. Krok symulacji opiera się na przeprowadzeniu przykładowej losowej rozgrywki z danego kroku. Podczas symulacji podejmowane są hipotetyczne decyzje oparte na wiedzy na temat gry i założeniu, że każdy gracz podejmuje dla siebie najlepszą decyzję. Przygotowałem funkcję oceny wartości danej karty. Ocena dokonywana jest na podstawie liczności koloru danej karty i jej wartości punktowej. Karty akcji poza posiadaną wyższą wartością punktowa posiadają obliczany współczynnik agresji na podstawie prawdopodobieństwa odbicia ataku na gracza. Oraz czynnika losowego opartego na liczbie pseudo losowej rozkładu normalnego.

Klasyfikacja najlepszego wyboru:

Propagacja wsteczna to krok występujący po zakończeniu symulacji. Aktualizuje dane na temat ilości i wyniku symulacji. Jest to krok służący do zmiany współczynników wykorzystywanych przy wyborze.

# Schemat blokowy algorytmu

Wszystkie diagramy czynności zostały przygotowane zgodnie z standardami Unified Modeling Language (UML)

Diagram 1 Schemat przeszukiwania drzewa metodami Monte Carlo

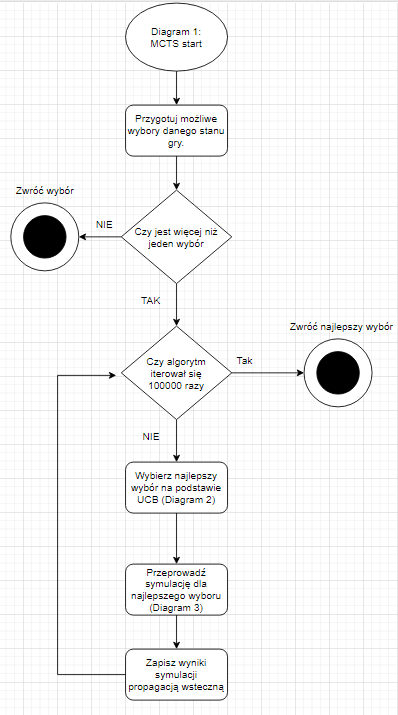
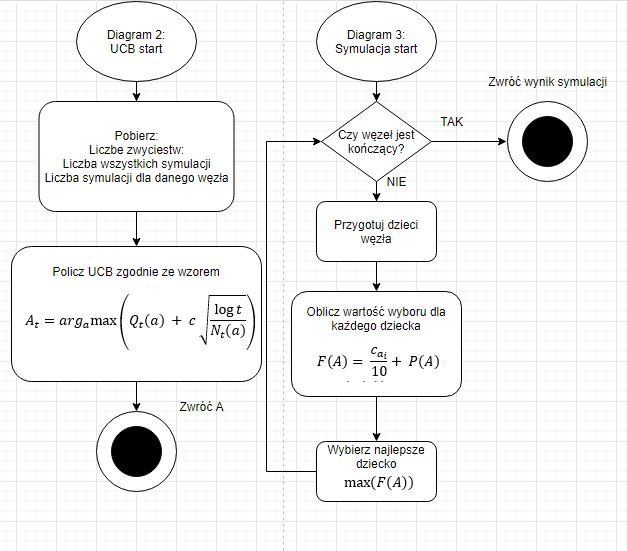


Diagram 2 Wyznaczanie najlepszego zlementu za pomocą UCB

Diagram 3 Przebieg symulacji dla przeszukiwania drzewa metodami Monte Carlo



# Opis programu

**Opis technologii**

**Java 8 – Obiektowy język wysokopoziomowy jest wykonywany przez JVM (Java Virtual Machine). Wersja 8 obsługiwana przez niemal wszystkie istniejące wirtualne maszyny.**

**JVM – wirtualna maszyna Javy, wykonuje skompilowany kod Javy.**

**JavaFX – Jest to framework przeznaczony do tworzenia okienkowych programów Javy.**

**Zaletą wybranych technologii jest ich uniwersalność. Program napisany w Java będzie działał tak samo na każdym komputerze z dostępną JVM.**

**Opis struktury programu**

**Całość została napisana w formacie Model View Presenter**

**View – Opis interfejsu użytkownika w formacie fxml.**

**Model – Logiczne elementy odpowiedzialne za przeprowadzenie rozgrywki.**

**Presenter - Element pośredniczący w komunikacji pomiędzy widokami(View), a modelami**

**Diagramy klas.**

**Dałość została podzielona na 3 części z elementem wspólnym w postaci GameEngineV2**

Diagram Klas 1 Część odpowiedzialna za komunikację z użytkownikiem

**Obraz zawierający stół

Opis wygenerowany automatycznie**

Diagram Klas 2 Część zawierająca elementy rozgrywki

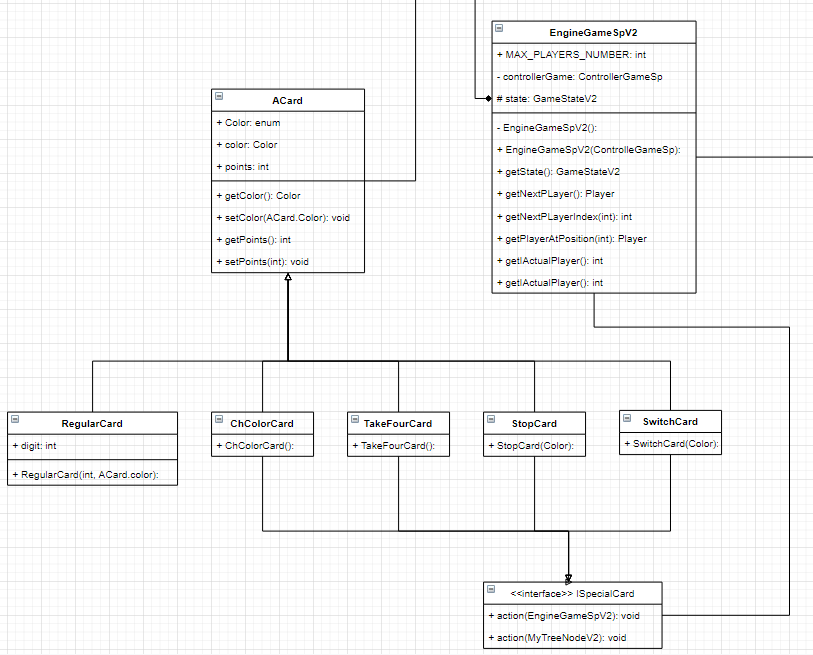
****

Diagram Klas 3 Część odpowiedzialna za realizację sztucznej inteligencji w grze

**Obraz zawierający stół

Opis wygenerowany automatycznie**

**Opis pól i publicznych metod algorytmu przeszukiwania drzewa metodami Monte Carlo**

|  |  |
| --- | --- |
| Klasa GameState – przechowuje dane dotyczące stanu rozgrywki | |
| Typ i nazwa pola | Opis pola |
| Vector<Players> players | Przechowuje obiekty graczy biorących udział w grze |
| int lastPlayerIndex | Indeks ostatniego gracza pokazuje liczbę graczy w rozgrywce |
| boolean isFrozen | Określenie czy następny gracz jest zamrożony |
| Stack<ACard> table | Stos stołu na który odkładane są karty |
| Vector<ACard> cardSet | Zbiór kart |
| Int[] playersHandsSizes | Rozmiary rąk graczy |
| Direction direction | Kierunek przebiegu gry |
| int actualPlayerIndex | Indeks aktualnie grającego gracza |
| int numberOfTakenCards | Liczba kart do dobrania dla następnego gracza |
| Typ i nazwa metody | Opis metody |
| GameStateV2 deepClone | Tworzy klona siebie i zwraca do niego referencje |
| void nextPlayer | Przestawia gracza na następnego |
| int getNextPlayerIndex | Pobiera indeks następnego gracza |
| void switchDirection | Zmienia kierunek przebiegu gry na przeciwny |
| void setNumberOfTakenCards | Ustawia liczbę kart do dobrania |

|  |  |
| --- | --- |
| Klasa AIPlayer rozszerzająca klasę player Player | |
| Typ i nazwa pola | Opis pola |
| int myRedCards | Liczba czerwonych kart w ręce |
| int myBlueCards | Liczba niebieskich kart w ręce |
| int myGreenCards | Liczba zielonych kart w ręce |
| int myYellowCards | Liczba żółtych kart w ręce |
| Vector<ACard> matchingCards | Wektor pasujących kart |
| GameStateV2 state | Referencja do stanu gry |
| Typ i nazwa metody | Opis metody |
| ACard.Color maxColorQuantity | Zwraca najliczniejszy kolor kart w ręku |
| boolean matchMyCards | Zwraca true gdy jest możliwość rzucenia choć jednej karty. |
| ACard MCTS | Tworzy drzewo Monte Carlo i zwraca najlepszą karte do położenia na stół |

|  |  |
| --- | --- |
| Klasa MyTreeMonteCarlo – implementuje logikę przeszukiwania drzewa MonteCarlo | |
| Typ i nazwa pola | Opis pola |
| Int ITERATIONS | Liczba iteracji symulacji |
| Random randomGenerator | Generator liczb pseudo-losowych rozkładu normalnego |
| MyTreeNodeV2 head | Node który jest aktualnym stanem gry jest to pierwszy węzeł drzewa MCTS |
| Typ i nazwa metody | Opis metody |
| ACard search | Uruchamia algorytm przeszukiwania drzewa Monte Carlo |
| MyTreeNodeV2 seleciton | Wybór następnego węzła do symulacji na podstawie UCB |
| double calcUCB | Wylicza współczynnik UCB |
| int simulation | Porusza się zgodnie z algorytmem |
| MyTreeNodeV2 randomPlay | Losowe zagranie podczas symulacji |
| double evaluate | Obliczanie wartości następnego ruchu w symulacji |
| void backpropagation | Propagacja wsteczna |

|  |  |
| --- | --- |
| Klasa MyTreeNodeV2 – implementuje logikę węzłów decyzji drzewa | |
| Typ i nazwa pola | Opis pola |
| Boolean isTerminal | Czy węzeł jest kończący |
| int visitCount | Licznik liczby symulacji |
| int winCount | Licznik zwycięstw symulacji |
| MyTreeNodeV2 parent | Ojciec węzła |
| GameStateV2 state | Stan gry |
| ACard[] failedCard | Tablica przechowuje karty graczy dla których dobrali kartę z deku. |
| ACard.Color topColor | Kolor karty na szczycie stosu kart na stole |
| Vector<Acard> myHand | Wektor ręki kart gracza symulącego |
| ArrayList<Vector<ACard>> playersHands | Lista rąk kart wszystkich graczy w grze. |
| Typ i nazawa metody | Opis metody |
| Void determine | Tworzy losowy hipotetyczny stan gry, bazując na jawnych danych |
| Vector<ACard> createChildren | Tworzy węzły dzieci dla danego węzła |
| Vector<ACard> getActualPlayerHand | Zwraca rękę kart aktywnego gracza dla węzła |
| MyTreeNodeV2 clone | Metoda klonuje obiekt MyTreeNodeV2 |

# Kod algorytmu

Klasa GameStateV2

package game.myAssets;  
  
import game.myAssets.cards.ACard;  
  
import java.util.Stack;  
import java.util.Vector;  
  
public class GameStateV2  
{  
 //Wektor graczy  
 Vector<Player> players;  
 //Index ostatniego gracza  
 public int lastPlayerIndex;  
 //..............  
 public boolean isFrozen;  
 //Stos stolu na ktory odkladane sa karty  
 public Stack<ACard> table;  
 //Zbior kart uzytych w grze  
 public Vector<ACard> cardSet;  
 //Kupka kart do gry  
 public Vector<ACard> deck;  
  
 //Rozmiary rak graczy  
 public int[] playersHandsSizes;  
 //Kierunki rozgrywki  
 public enum Direction  
 {  
 *CLOCKWISE*,  
 *COUNTERCLOCKWISE* }  
 public Direction direction = Direction.*CLOCKWISE*;  
 public int actualPlayerIndex = 0;  
 public int numberOfTakenCards = 0;  
  
 public GameStateV2()  
 {  
 this.players = new Vector<>();  
 this.table = new Stack<>();  
 this.cardSet = new Vector<>();  
 this.deck = new Vector<>();  
 }  
  
 public GameStateV2 deepClone()  
 {  
 GameStateV2 tmpState = new GameStateV2();  
 tmpState.players = (Vector<Player>) this.players.clone();  
 tmpState.lastPlayerIndex = this.lastPlayerIndex;  
 tmpState.table = (Stack<ACard>)this.table.clone();  
 tmpState.deck = (Vector<ACard>)this.deck.clone();  
 tmpState.playersHandsSizes = new int[this.players.size()];  
 for(int i = 0; i < this.players.size(); ++i)  
 {  
 tmpState.playersHandsSizes[i] = this.playersHandsSizes[i];  
 }  
 tmpState.cardSet = (Vector<ACard>)this.cardSet.clone();  
 tmpState.direction = this.direction;  
 tmpState.actualPlayerIndex = this.actualPlayerIndex;  
 tmpState.numberOfTakenCards = this.numberOfTakenCards;  
 return tmpState;  
 }  
  
 */\*\*  
 \* Metoda przesowa index o jeden  
 \*/* public void nextPlayer()  
 {  
 if(direction == Direction.*CLOCKWISE*)  
 {  
 if(actualPlayerIndex == lastPlayerIndex)  
 actualPlayerIndex = 0;  
 else  
 actualPlayerIndex++;  
 }  
 else  
 {  
 if (actualPlayerIndex == 0)  
 actualPlayerIndex = lastPlayerIndex;  
 else  
 actualPlayerIndex--;  
 }  
 if(isFrozen)  
 {  
 isFrozen = false;  
 if(direction == Direction.*CLOCKWISE*)  
 {  
 if(actualPlayerIndex == lastPlayerIndex)  
 actualPlayerIndex = 0;  
 else  
 actualPlayerIndex++;  
 }  
 else  
 {  
 if (actualPlayerIndex == 0)  
 actualPlayerIndex = lastPlayerIndex;  
 else  
 actualPlayerIndex--;  
 }  
 }  
 }  
  
 public int getNextPlayerIndex()  
 {  
 if(direction == Direction.*CLOCKWISE*)  
 {  
 if(actualPlayerIndex == lastPlayerIndex)  
 return 0;  
 else  
 return actualPlayerIndex + 1;  
 }  
 else  
 {  
 if (actualPlayerIndex == 0)  
 return lastPlayerIndex;  
 else  
 return actualPlayerIndex - 1;  
 }  
 }  
 */\*\*  
 \* zmien kierunek  
 \*/* public void switchDirection()  
 {  
 if(this.direction == GameStateV2.Direction.*CLOCKWISE*)  
 this.direction = GameStateV2.Direction.*COUNTERCLOCKWISE*;  
 else  
 this.direction = GameStateV2.Direction.*CLOCKWISE*;  
 }  
  
 */\*\*  
 \* ustaw liczbe pobieranych kart  
 \*/* public void setNumberOfTakenCards(int numberOfTakenCards)  
 {  
 this.numberOfTakenCards = numberOfTakenCards;  
 }  
}

Klasa MyTreeMonteCarlo

package game.myAssets.AI;  
  
import game.myAssets.GameStateV2;  
import game.myAssets.cards.ACard;  
import game.myAssets.cards.ISpecialCard;  
import game.myAssets.cards.TakeTwoCard;  
  
import java.util.Random;  
import java.util.Vector;  
  
public class MyTreeMonteCarlo  
{  
 private final int ITERATIONS = 100000;  
  
 private Random randomGenerator = new Random();  
 MyTreeNodeV2 head;  
  
 */\*\*  
 \* Konstruktor tworzy obiekt obslugujacy wyszukiwanie monte carlo  
 \** ***@param*** *state GameState - Stan gry  
 \** ***@param*** *myHand Vector<ACard> reka z kartami  
 \*/* public MyTreeMonteCarlo(GameStateV2 state, Vector<ACard> myHand)  
 {  
 this.head = new MyTreeNodeV2(state, (Vector<ACard>) myHand.clone());  
 }  
  
 */\*\*  
 \* Zwraca znaleziona karte do rzucenia  
 \** ***@return*** *ACard - znaleziona karta lub  
 \** ***@return*** *null - w przypadku braku dopasowania  
 \*/* public ACard search()  
 {  
 head.createChildren();  
 for(int i = 0; i < ITERATIONS; ++i)  
 {  
 MyTreeNodeV2 nodeToExploration = selection(head);  
 //if(nodeToExploration.isTerminal)  
 //return nodeToExploration.state.table.peek();  
 nodeToExploration = nodeToExploration.clone();  
 nodeToExploration.determine();  
 int result = simulation(nodeToExploration);  
 backpropagation(nodeToExploration, result);  
 }  
 System.*out*.println("Liczbna dzieci: " + head.children.size());  
 for (MyTreeNodeV2 child:  
 head.children)  
 {  
 System.*out*.println("Liczba odwiedzeni: " + child.visitCount + "\tLiczba zwyciestw: " + child.winCount);  
 }  
 MyTreeNodeV2 bestNode = selection(head);  
 return bestNode.state.table.peek();  
 }  
  
 */\*\*  
 \* Metoda dokonuje wyboru na podstawie wskaznika UCT z posrod wezlow dzieci podanego wezla  
 \** ***@param*** *node  
 \** ***@return*** *\*/* public MyTreeNodeV2 selection(MyTreeNodeV2 node)  
 {  
 Vector<MyTreeNodeV2> children = node.children;  
 MyTreeNodeV2 bestMove = children.get(0);  
 double bestUTC = Integer.*MIN\_VALUE*;  
 if(children.size() > 0)  
 {  
 for (MyTreeNodeV2 child: children)  
 {  
 int parentVisitCount = node.visitCount;  
 int childVisitCount = child.visitCount;  
 int win = child.winCount;  
 double calcedUTC = calcUCB(win, parentVisitCount, childVisitCount);  
 if(calcedUTC > bestUTC)  
 {  
 bestUTC = calcedUTC;  
 bestMove = child;  
 }  
 }  
 return bestMove;  
 }  
 else  
 return null;  
 }  
  
 */\*\*  
 \* winCount is reward value of all nodes beneath this node  
 \* N is the number of times the parent node has been simulated, and  
 \* ni is the number of times the child node i has been visited  
 \* \*/* private double calcUCB(double winCount, int bigN, int ni)  
 {  
 if(ni == 0)  
 {  
 return Integer.*MAX\_VALUE*;  
 }  
 double c = Math.*sqrt*(2.0); //state parameter  
 double rootBase = Math.*log*(bigN) / ni;  
 return winCount / ni + c \* Math.*sqrt*(rootBase);  
 }  
  
 */\*\*  
 \*  
 \** ***@param*** *node  
 \** ***@return*** *\*/* public int simulation(MyTreeNodeV2 node)  
 {  
 MyTreeNodeV2 tmpNode = node;  
 while(tmpNode.isTerminal == false)  
 {  
 tmpNode = randomPlay(tmpNode);  
 }  
 if(tmpNode.playersHands.get(head.state.actualPlayerIndex).size() == 0)  
 {  
 return 1;  
 }else if(tmpNode.state.cardSet.size() == 0)  
 {  
 return 0;  
 }else  
 {  
 return -1;  
 }  
 }  
  
 */\*\*  
 \* Metoda wykorzystuje ekspercka wiedze gry w celu wybierania losowych ruchow symulacji  
 \** ***@param*** *node  
 \** ***@return*** *\*/* public MyTreeNodeV2 randomPlay(MyTreeNodeV2 node)  
 {  
 double lowerbound = -0.5;  
 node.createChildren();  
 int bestIndex = 0;  
 double evaluation = Integer.*MIN\_VALUE*;  
 Vector<MyTreeNodeV2> children = node.children;  
 if(children.size() > 1)  
 {  
 for (int i = 0; i < children.size(); ++i)  
 {  
 double result = evaluate(children.get(i));  
 result += randomGenerator.nextDouble() + lowerbound;  
 if (result > evaluation)  
 {  
 evaluation = result;  
 bestIndex = i;  
 }  
 }  
 }  
 MyTreeNodeV2 randomChoice = children.get(bestIndex);  
 return randomChoice;  
 }  
  
 */\*\*  
 \* Metoda dokonuje wyboru na podstawie zmiennej losowej i aktualnego stanu gry i reki aktualnego gracza  
 \** ***@return*** *metoda zwraca wartosc wyboru  
 \* \*/* public double evaluate(MyTreeNodeV2 myTreeNode)  
 {  
 double cardValue = 0.0;  
 GameStateV2 state = myTreeNode.state;  
 ACard card = state.table.peek();  
 //Licznosc koloru kary  
 switch (card.getColor())  
 {  
 case *RED*:  
 cardValue += myTreeNode.myRedCards / 10;  
 break;  
 case *BLUE*:  
 cardValue += myTreeNode.myBlueCards / 10;  
 break;  
 case *GREEN*:  
 cardValue += myTreeNode.myGreenCards / 10;  
 break;  
 case *YELLOW*:  
 cardValue += myTreeNode.myYellowCards / 10;  
 default:  
 break;  
 }  
 if(card instanceof ISpecialCard)  
 cardValue += 0.5;  
 //Liczy prawdopodobienstwo odbicia  
 if(card instanceof TakeTwoCard && myTreeNode.remainingTakeTwoCards > 0)  
 {  
 double N = 54 - myTreeNode.state.table.size();  
 double K = myTreeNode.remainingTakeTwoCards;  
 int nextPlayerIndex = myTreeNode.state.getNextPlayerIndex();  
 double n = myTreeNode.playersHands.get(nextPlayerIndex).size();  
 cardValue += cutFactorial(N, N-K) / cutFactorial(N-n, N-n-K);  
 }  
 //Czy nastepny gracz nie mogl rzucic karty  
 if(myTreeNode.failedCard[state.actualPlayerIndex] != null && card.getColor() == myTreeNode.failedCard[state.actualPlayerIndex].getColor())  
 cardValue += 1.0;  
 return cardValue;  
 }  
  
 private double cutFactorial(double firsNum, double secNum)  
 {  
 double result = 1;  
 if(secNum > firsNum)  
 {  
 double tmp = secNum;  
 secNum = firsNum;  
 firsNum = tmp;  
 }  
 for(double i = firsNum; i > secNum; i -= 0.1)  
 result \*= i;  
 return result;  
 }  
//\*/  
 */\*\*  
 \*  
 \** ***@param*** *nodeToExplore  
 \** ***@param*** *simulationResult  
 \*/* public void backpropagation(MyTreeNodeV2 nodeToExplore, int simulationResult)  
 {  
 while(nodeToExplore != null)  
 {  
 nodeToExplore.visitCount++;  
 nodeToExplore.winCount += simulationResult;  
 nodeToExplore = nodeToExplore.parent;  
 }  
 }  
}

klasa MyTreeNodeV2

package game.myAssets.AI;  
  
import game.myAssets.GameStateV2;  
import game.myAssets.cards.ACard;  
import game.myAssets.cards.ISpecialCard;  
import game.myAssets.cards.RegularCard;  
import game.myAssets.cards.TakeTwoCard;  
  
import java.util.ArrayList;  
import java.util.Collections;  
import java.util.Vector;  
  
public class MyTreeNodeV2  
{  
 public boolean isTerminal = false;  
 public int visitCount = 0;  
 public int winCount = 0;  
 MyTreeNodeV2 parent = null;  
 GameStateV2 state;  
 ACard[] failedCard;  
 ACard.Color topColor;  
 Vector<ACard> myHand;  
 ArrayList<Vector<ACard>> playersHands = new ArrayList<>();  
 Vector<MyTreeNodeV2> children = new Vector<>();  
  
 int myRedCards = 0;  
 int myBlueCards = 0;  
 int myGreenCards = 0;  
 int myYellowCards = 0;  
 int myBlackCards = 0;  
  
 int remainingTakeTwoCards = 8;  
  
 */\*\*  
 \* Konstruktor podstawowy tworzy wezel aktualnego stanu gry  
 \** ***@param*** *state  
 \*/* MyTreeNodeV2(GameStateV2 state, Vector<ACard> myHand)  
 {  
 this.state = state;  
 this.myHand = myHand;  
 this.topColor = state.table.peek().getColor();  
 this.failedCard = new ACard[4];  
 for (ACard card: this.myHand)  
 {  
 myColorCardIncrement(card);  
 }  
 }  
  
 MyTreeNodeV2(MyTreeNodeV2 parentNode)  
 {  
 this.isTerminal = parentNode.isTerminal;  
 this.visitCount = parentNode.visitCount;  
 this.winCount = parentNode.winCount;  
 this.parent = parentNode;  
 this.state = parent.state.deepClone();  
 this.failedCard = new ACard[4];  
 for (int i = 0; i <= state.lastPlayerIndex; ++i)  
 {  
 failedCard[i] = parent.failedCard[i];  
 }  
 this.topColor = parent.topColor;  
 this.myHand = (Vector<ACard>) parent.myHand.clone();  
 for (Vector<ACard> hand: parent.playersHands)  
 {  
 this.playersHands.add((Vector<ACard>)hand.clone());  
 }  
 for (ACard card: this.myHand)  
 {  
 myColorCardIncrement(card);  
 }  
 this.remainingTakeTwoCards = parentNode.remainingTakeTwoCards;  
 if(state.table.peek() instanceof TakeTwoCard)  
 this.remainingTakeTwoCards--;  
 }  
  
 public GameStateV2 getState()  
 {  
 return state;  
 }  
  
 public ACard.Color getTopColor()  
 {  
 return topColor;  
 }  
  
 public void setTopColor(ACard.Color topColor)  
 {  
 this.topColor = topColor;  
 }  
  
 */\*\*  
 \* Metoda generuje losowy stan gdy na podstawie posiadanych informacji  
 \*/* public void determine()  
 {  
 Collections.*shuffle*(state.cardSet);  
 for (int i = 0; i < state.playersHandsSizes.length; ++i)  
 {  
 if(i == state.actualPlayerIndex)  
 playersHands.add(myHand);  
 else  
 {  
 Vector<ACard> deterministicOpponentHand = new Vector<>();  
 while (deterministicOpponentHand.size() < state.playersHandsSizes[i])  
 {  
 deterministicOpponentHand.add(state.cardSet.remove(0));  
 }  
 playersHands.add(deterministicOpponentHand);  
 }  
 }  
 }  
  
 */\*\*  
 \* Listuje mozliwe do wykonania ruchy w danej turze  
 \** ***@return*** *\*/* public ArrayList<ACard> getPossibleMoves()  
 {  
 Vector<ACard> actualPlayerHand = getActualPlayerHand();  
 ArrayList<ACard> matchingCards = new ArrayList<ACard>();  
 if(state.table.peek() instanceof ISpecialCard)  
 {  
 for (ACard card: actualPlayerHand)  
 {  
 if(card.getColor() == topColor || card.getColor() == ACard.Color.*BLACK*)  
 matchingCards.add(card);  
 }  
 }  
 else if(state.table.peek() instanceof RegularCard)  
 {  
 for (ACard card: actualPlayerHand)  
 {  
 if(card instanceof RegularCard &&  
 (((RegularCard) card).getDigit() == ((RegularCard)state.table.peek()).getDigit() ||  
 card.getColor() == topColor ||  
 card.getColor() == ACard.Color.*BLACK*))  
 {  
 matchingCards.add(card);  
 }  
 }  
 }  
 return matchingCards;  
 }  
  
 */\*\*  
 \* Tworzy dzieci z mozliwych ruchow zwroconych przez metode getPossibleMoves()  
 \* Gdy nie jest w stanie wykonac ruchu to tworzy dzicko w ktorym dobiera jedna karte  
 \* Klonuje swoj stan natepnie na stworzonym klonie dokonuje zmian stanu  
 \** ***@return*** *wektor stworzonych dzieci  
 \*/* public Vector<MyTreeNodeV2> createChildren()  
 {  
 children.clear();  
 ArrayList<ACard> possibleMoves = getPossibleMoves();  
 if(possibleMoves.size() > 0)  
 {  
 for (ACard possibleMove: possibleMoves)  
 {  
 MyTreeNodeV2 childNode = new MyTreeNodeV2(this);  
 childNode.cardOnTable(possibleMove);  
 childNode.state.nextPlayer();  
 if(childNode.state.numberOfTakenCards > 0)  
 {  
 childNode.drawMany();  
 childNode.state.nextPlayer();  
 }  
 children.add(childNode);  
 }  
 }  
 else  
 {  
 MyTreeNodeV2 childNode = new MyTreeNodeV2(this);  
 childNode.drawOne();  
 childNode.state.nextPlayer();  
 children.add(childNode);  
 }  
 return this.children;  
 }  
  
 */\*\*  
 \* Usuwa karte z reki odpowiedniego gracza bazujac na isMaxPlayer  
 \* Nastepnie przeklada wybrana karte na stol  
 \* \*/* private void cardOnTable(ACard card)  
 {  
 if (card == null) throw new AssertionError();  
 Vector<ACard> actualPlayerHand = getActualPlayerHand();  
 actualPlayerHand.remove(card);  
 state.table.add(card);  
 if(card instanceof ISpecialCard)  
 ((ISpecialCard)card).action(this);  
 state.playersHandsSizes[state.actualPlayerIndex] -= 1;  
 if(state.playersHandsSizes[state.actualPlayerIndex] <= 0)  
 this.isTerminal = true;  
 if(card.getColor() != ACard.Color.*BLACK*)  
 topColor = card.getColor();  
 failedCard[state.actualPlayerIndex] = null;  
 myColorCardDecrement(card);  
 }  
  
 */\*\*  
 \* Metoda odpowiada za symulacje dobrania jednej karty w przypadku braku mozliwosci znalezienia odpowiedniego wyboru  
 \* Ustawia flage failedCard  
 \* \*/* private void drawOne()  
 {  
 if(state.cardSet.size() == 0) return;  
 failedCard[state.actualPlayerIndex] = state.table.peek();  
 ACard takenCard = state.cardSet.remove(0);  
 if(takenCard instanceof ISpecialCard)  
 {  
 if(takenCard.getColor() == ACard.Color.*BLACK*)  
 {  
 cardOnTable(takenCard);  
 return;  
 }  
 else if(takenCard.getClass() == state.table.peek().getClass())  
 {  
 cardOnTable(takenCard);  
 return;  
 }  
 else if(takenCard.getColor() == state.table.peek().getColor())  
 {  
 cardOnTable(takenCard);  
 return;  
 }  
 }  
 else  
 {  
 if (takenCard.getClass() == state.table.peek().getClass())  
 {  
 if (takenCard.getColor() == state.table.peek().getColor()  
 || ((RegularCard) takenCard).getDigit() == ((RegularCard) state.table.peek()).getDigit())  
 {  
 cardOnTable(takenCard);  
 return;  
 }  
 }else if(takenCard.getColor() == state.table.peek().getColor())  
 {  
 state.table.add(takenCard);  
 if(takenCard instanceof ISpecialCard)  
 ((ISpecialCard)takenCard).action(this);  
 return;  
 }  
 }  
 getActualPlayerHand().add(takenCard);  
 failedCard[state.actualPlayerIndex] = state.table.peek();  
 myColorCardIncrement(takenCard);  
 }  
  
 private void drawMany()  
 {  
 if(state.numberOfTakenCards <= 0) return;  
 if(this.playersHands.isEmpty()) return; // wyjatek  
 ACard firstCard = state.cardSet.remove(0);  
 //Jesli jest posiadana  
  
 if(state.table.peek() instanceof TakeTwoCard &&  
 firstCard instanceof TakeTwoCard)  
 {  
 state.table.add(firstCard);  
 ((ISpecialCard)firstCard).action(this);  
 return;  
 }  
 Vector<ACard> hand = getActualPlayerHand();  
 hand.add(firstCard);  
 myColorCardIncrement(firstCard);  
 state.numberOfTakenCards--;  
 while (state.numberOfTakenCards > 0)  
 {  
 ACard card = state.cardSet.remove(0);  
 hand.add(card);  
 myColorCardIncrement(card);  
 state.numberOfTakenCards--;  
 }  
 }  
  
 */\*\*  
 \* Metoda zwraca reke aktualnego gracza  
 \* Przed determinizacja zostaje zwrocona reka gracza  
 \** ***@return*** *\*/* public Vector<ACard> getActualPlayerHand()  
 {  
 if(playersHands.size() == 0)  
 return myHand;  
 else  
 return playersHands.get(state.actualPlayerIndex);  
 }  
  
 */\*\*  
 \* Metoda klonuje obiekt MyTreeNodeV2  
 \** ***@return*** *\*/* public MyTreeNodeV2 clone()  
 {  
 MyTreeNodeV2 clonedNode = new MyTreeNodeV2(this);  
 return clonedNode;  
 }  
  
 public ACard.Color getActualPlayerBestColor()  
 {  
 int numberOfRed = 0;  
 int numberOfBlue = 0;  
 int numberOfGreen = 0;  
 int numberOfYellow = 0;  
 for (ACard card: getActualPlayerHand())  
 {  
 switch (card.getColor())  
 {  
 case *RED*:  
 numberOfRed++;  
 break;  
 case *BLUE*:  
 numberOfBlue++;  
 break;  
 case *GREEN*:  
 numberOfGreen++;  
 break;  
 case *YELLOW*:  
 numberOfYellow++;  
 break;  
 }  
 }  
 if(numberOfRed >= numberOfBlue && numberOfRed >= numberOfGreen && numberOfRed >= numberOfYellow)  
 return ACard.Color.*RED*;  
 else if(numberOfBlue >= numberOfGreen && numberOfBlue >= numberOfYellow)  
 return ACard.Color.*BLUE*;  
 else if(numberOfGreen >= numberOfYellow)  
 return ACard.Color.*GREEN*;  
 else  
 return ACard.Color.*YELLOW*;  
 }  
  
 private void myColorCardIncrement(ACard card)  
 {  
 switch (card.getColor())  
 {  
 case *RED*:  
 myRedCards++;  
 break;  
 case *BLUE*:  
 myBlueCards++;  
 break;  
 case *GREEN*:  
 myGreenCards++;  
 break;  
 case *YELLOW*:  
 myYellowCards++;  
 break;  
 case *BLACK*:  
 myBlackCards++;  
 break;  
 }  
 }  
  
 private void myColorCardDecrement(ACard card)  
 {  
 switch (card.getColor())  
 {  
 case *RED*:  
 myRedCards--;  
 break;  
 case *BLUE*:  
 myBlueCards--;  
 break;  
 case *GREEN*:  
 myGreenCards--;  
 break;  
 case *YELLOW*:  
 myYellowCards--;  
 break;  
 case *BLACK*:  
 myBlackCards--;  
 break;  
 }  
 }  
}

# Instalacja programu

1. **Instalacja Javy. Należy pobrać Javę z** [Download Java for Windows](https://java.com/pl/download/) . Jeżeli Java jest już zainstalowana to należy ją zaktualizować
2. **Uruchomienie UNO**.jar klikając dwukrotnie

Uwaga – folder assets powinien znajdować się w tym samym miejscu co UNO.jar w celu poprawnego działania gry

# Zawartość folderu zip

Dokumentacja

Uno.jar

Folder sprites

# Instrukcja obsługi programu

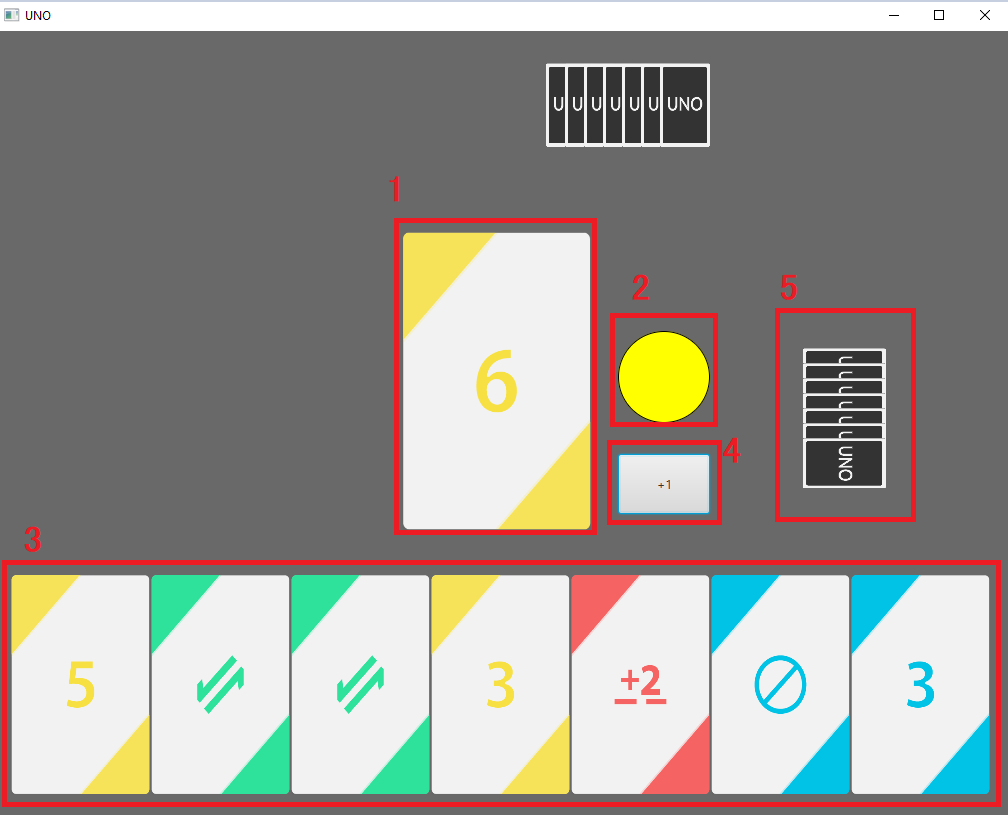
Menu główne i interfejs użytkownika

Obraz zawierający tekst

Opis wygenerowany automatycznie

1. **„Rozgrywka dla jednego gracza” uruchamia rozgrywkę w trybie dla pojedynczego gracza z sztuczną inteligencją.**
2. **„Rozgrywka dla wielu graczy” uruchamia rozgrywkę w trybie dla wielu graczy z opcjonalną sztuczną inteligencją w grze.**

**Ekran gry**

****

1. **Ostatnia karta na stosie. Pokazuje do jakiej karty gracz musi się dopasować**
2. **Wskaźnik koloru karty**
3. **Ręka gracza, znajdują się w niej karty jakie gracz posiada w ręce. Kliknięcie na kartę powoduje jej wybór. Po wyłożeniu poprawnej karty na stos tura gracza kończy się**
4. **Przycisk doboru karty. Po kliknięciu zostaje dobrana jedna karta po potwierdzeniu przyjęcia karty tura gracza kończy się.**
5. **Ręka przeciwnika**
6. **Po wyłożeniu ostatniej karty gra automatycznie kończy się i wraca do menu głównego.**

# Bibliografia

Monte Carlo Tree Search for games with Hidden Information and Uncertainty - Daniel Whitehouse PhD University of York Computer Science August, 2014

<https://core.ac.uk/download/pdf/30267707.pdf>

Information Set Monte Carlo Tree Search - Peter I. Cowling, Member, IEEE, Edward J. Powley, Member, IEEE, and Daniel Whitehouse, Student Member, IEEE

<https://pure.york.ac.uk/portal/files/13014166/CowlingPowleyWhitehouse2012.pdf>

Comparison of Monte Carlo Tree Search Methods in the Imperfect Information Card Game Cribbage -Richard Kelly and David Churchill Computer Science Faculty of Science Memorial University {richard.kelly, [dchurchill}@mun.ca](mailto:dchurchill%7d@mun.ca)

<http://www.cs.mun.ca/~dchurchill/pdf/necec17cribbage.pdf>

Upper Confidence Bound Algorithm in Reinforcement Learning - GeeksforGeeks

<https://www.geeksforgeeks.org/upper-confidence-bound-algorithm-in-reinforcement-learning/>

Re-determinizing MCTS in Hanabi - James Goodman james@janigo.co.uk

<https://ieee-cog.org/2019/papers/paper_17.pdf>

# Oświadczenia

Obraz zawierający tekst

Opis wygenerowany automatycznie