Uniwersytet
Kardynała Stefana Wyszyńskiego
w Warszawie

Projekt „Kierki”

Autor: Marta Zaorska

Warszawa, 03.06.2023

Spis treści

Wstęp

Dany dokument stanowi opis projektu informatycznego, wykonanego w ramach przedmiotu „Program programistyczny indywidualny”.

Celem realizacji projektu była implementacja gry karcianej „Kierki”, przeznaczonej dla jednego użytkownika. Użytkownik ma możliwość wyboru progu punktowego dla danej rozgrywki. Główną ideą projektu było utworzenie algorytmu dokonującego wyboru karty (symulacji) dla pozostałych trzech graczy.

Istotnym założeniem było utworzenie progresywnej aplikacji webowej (PWA). Jej celem jest zapewnienie użytkownikowi możliwości korzystania z aplikacji w trybie offline oraz możliwość pobrania jej na urządzenia mobilne oraz na desktop.

Zasady gry

„Kierki” to gra przeznaczona dla 4 graczy. Do gry wykorzystana jest cała talia kart (52 karty, bez Jokerów), rozdana równo pomiędzy graczy. Po rozdaniu każdy gracz wybiera trzy swoje karty i przekazuje je innemu graczowi (gracz musi wybrać karty do przekazania przed obejrzeniem kart, które sam dostał). W pierwszym, drugim i trzecim rozdaniu gracz przekazuje karty pozostałym graczom. W czwartym rozdaniu gracze nie przekazują sobie kart. Taki cykl powtarza się do końca gry.

Grę rozpoczyna gracz, który posiada dwójkę trefl (rozpoczyna pierwszą lewę). Każdą lewę zdobywa gracz dający najwyższą kartę tego samego koloru co karta rozpoczynająca lewę. Należy grać do koloru. Jeśli gracz nie ma karty pasującej kolorem, to może rzucić kartę dowolnego koloru.

Gracz nie może rozpocząć lewy od damy pik lub kiera, jeżeli żaden z graczy nie dodał już kiera (do wcześniejszej lewy). Wyjątek stanowi sytuacja, w której gracz nie ma możliwości zagrania inną kartą.

Po zakończeniu rozdania graczom dopisuje się 1 punkt za zebranego kiera oraz 13 punktów graczowi, który zdobył damę pik. Wyjątkiem stanowi zdobycie premii – zebranie wszystkich punktowanych kart (wszystkie kiery i dama pik), wtedy wszystkim przeciwnikom dopisuje się po 26 punktów.

Jeżeli po zakończeniu rozdania dowolny gracz przekroczył określoną przez użytkownika liczbę punktów to gra jest zakończona i wygrywa osoba z najmniejszą liczbą punktów. W przeciwnym wypadku następuje kolejne rozdanie. (Kierki, 2021)

Charakterystyka modelu

Tasowanie kart – algorytm Fisher–Yates

Algorytm Fisher-Yates to algorytm tasowania skończonej sekwencji. W projekcie zastosowano nowoczesną wersję tego algorytmu o złożoności czasowej liniowej. Algorytm przetwarza sekwencję od końca, dla każdego elementu losuje inny element z nieprzetasowanej listy, następnie zamienia te dwa elementy miejscami. W wyniku przetwarzania na końcu tworzona jest lista z elementami już przetasowanymi. Do wybrania losowego elementu z sekwencji została użyta wbudowana funkcja w języku JavaScript *Math.random()* – generator liczb losowych z rozkładem jednostajnym. (Fisher–Yates shuffle, 2023)

Monte Carlo Tree Search

MCTS to algorytm wyszukiwania używany do podejmowania decyzji w grach z wysokimi czynnikami rozgałęzienia i niedoskonałymi informacjami. Łączy w sobie elementy symulacji losowej i przeszukiwania drzewa w celu zbudowania drzewa wyszukiwania reprezentującego różne możliwe działania i ich wyniki. Celem tego algorytmu jest wybór najlepszej akcji (wyboru karty) przez komputer.

Składowe algorytmu:

1. Inicjalizacja - utworzenie węzła głównego reprezentującego bieżący stan gry.
2. Selekcja - przechodzenie przez drzewo od węzła głównego (korzenia) do węzła liścia, w każdym węźle wybieramy węzeł podrzędny.
3. Rozszerzanie – dodawanie węzłów podrzędnych reprezentujących prawidłowe działania, które można wykonać z bieżącego stanu gry.
4. Symulacja - przeprowadzenie losowej rozgrywki z jednego z nowo rozwiniętych węzłów potomnych, aż do osiągnięcia stanu końcowego (końca rundy).
5. Propagacja wsteczna - aktualizacja statystyk (liczba wizyt i łączny wynik) dla wszystkich węzłów na ścieżce od wybranego węzła do węzła głównego.
6. Wybór najlepszej akcji - po żądanej liczbie iteracji wybieramy najlepszą akcję do wykonania z węzła głównego.

Proces selekcji węzła podrzędnego przeprowadzamy przy użyciu formuły UCT (Upper Confidence Bound for Trees), która uwzględnia statystyki węzła (liczbę wizyt, całkowity wynik) oraz stałą eksploracji (C).

$$UCT = \frac{child.score}{child.visits} + C * \sqrt{\frac{\ln(node.visits)}{child.visits}}$$

Wartość C wpływa na równowagę między odkrywaniem nowych ścieżek (węzłów) a wykorzystywaniem najbardziej obiecujących znalezionych do tej pory.

W projekcie do selekcji węzła podrzędnego w celu rozszerzania drzewa Monte Carlo zastosowano wartość współczynnika C = 2, aby położyć nacisk na odkrywanie nowych możliwości. Natomiast do wyboru najbardziej obiecującej akcji zastosowano wartość współczynnika C = 0. (Monte-Carlo Tree Search, 2023)

Opis programu

Środowisko

Aplikacja została utworzona przy użyciu programu Visual Studio Code. Projekt stworzony został w środowisku Node.js (wersja 14.17.2). Instalacja modułów i wtyczek za pomocą npm (node package manager, wersja 6.14.13).

Logika gry została zaimplementowana w skryptowym języku programowania wysokiego poziomu JavaScript.

Za warstwę prezentacyjną odpowiada język służący do opisu formy prezentacji stron internetowych CSS (Cascading Style Sheets) oraz język znaczników HTML.

Do konfiguracji aplikacji zastosowany został Webpack, którego głównym celem jest tworzenie pakietów, które uwzględnią zależności między modułami i klasami oraz poprawiają wydajność aplikacji.

W celu konfiguracji aplikacji za pomocą Webpack zostały wykorzystane poniższe moduły i wtyczki:

1. Moduły instalacyjne Webpack:
    - webpack – wersja 5.78.0
    - webpack-cli – wersja 5.0.1
    - webpack-dev-server – wersja 4.13.2

2. Transpilator JavaScript:
    - @babel/core – wersja 7.21.4
    - @babel/preset-env – wersja 7.21.4
    - babel-loader – wersja 9.1.2

3. Kompilacja i minifikacja plików CSS (warstwa prezentacyjna):
    - css-loader – wersja 6.7.3
    - style-loader – wersja 3.3.2

4. Konfiguracja pliku HTML:
    - html-webpack-plugin – wersja 5.5.0

5. Konfiguracja progresywnej aplikacji webowej (PWA):
    - workbox-webpack-plugin – wersja 6.5.4

6. Hostowanie aplikacji za pomocą Github Pages:
    - gh-pages – wersja 5.0.0

Struktura

Wbudowane w języku JavaScript struktury danych użyte w kodzie aplikacji to tablica oraz mapa (zbiór złożony z par klucz-wartość).

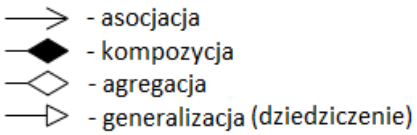
Za pomocą mapy (Map) zostało utworzone drzewo wyszukiwania dla algorytmu MCTS. Klucz stanowi krawędź w drzewie (akcja), natomiast wartością powiązaną z danym kluczem jest węzeł (obiekt klasy *Node)*, który posiada statystyki (wynik, liczbę wizyt) oraz stan gry po wykonaniu danej akcji.

Własne klasy:

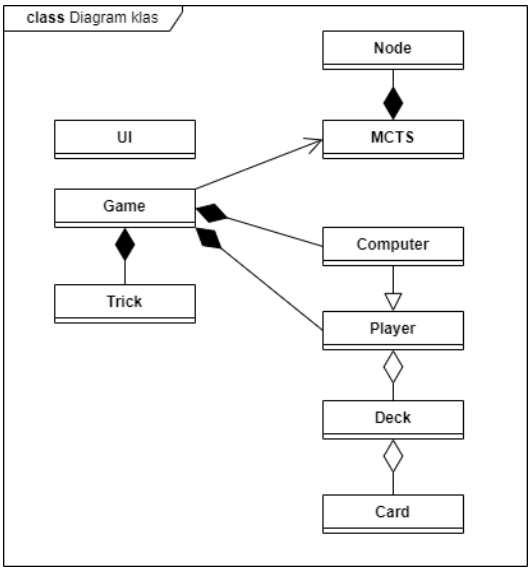
1. ***Deck*** - klasa reprezentująca zbiór kart, wykorzystuje tablicę do przechowywania obiektów klasy *Card.*
2. ***Card*** – klasa reprezentująca pojedynczą kartę, posiada właściwości wyróżniające daną kartę: kolor (ciąg tekstowy zawierający symbol koloru danej karty: '♣', '♦', '♠', '♥'), wartość (ciąg tekstowy reprezentujący figurę danej karty: ‘2’, ‘3’, ‘4’, ‘5’, ‘6’, ‘7’, ‘8’, ‘9’, ’10’, ‘J’, ‘Q’, ‘K’, ‘A’) oraz punkty (liczba całkowita reprezentująca wartość punktową danej karty: 0, 1 lub 13).
3. ***Trick*** – klasa reprezentująca lewę. Zawiera tablicę, która posiada wyłożone przez graczy karty w danej lewie oraz informację o kolorze danej lewy.
4. ***Player*** – klasa reprezentująca gracza. Przechowuje informację o punktach, które posiada gracz oraz obiekt klasy *Deck* reprezentujący karty posiadana przez gracza.
5. ***Computer*** – klasa pochodna reprezentująca gracza-komputer, która dziedziczy po klasie bazowej *Player.*
6. ***Game*** – klasa zawierająca całą logikę gry. Posiada funkcjonalność oraz właściwości reprezentujące aktualny stan gry.
7. ***Node*** – klasa reprezentująca pojedynczy węzeł w drzewie dla algorytmu MCTS. Posiada informację reprezentujące dany węzeł: statystyki, stan gry oraz właściwości *children* (obiekt Map) i *parent*.
8. ***MCTS*** – klasa zawierająca całą logikę Monte Carlo Tree Search**.**
9. ***UI*** – klasa zawierająca funkcjonalność przeznaczoną do zarządzania i manipulacji interfejsem użytkownika.

Poniższy diagram powstał przy użyciu https://app.diagrams.net/

Symbole:



Symbole określają związki pomiędzy klasami. Asocjacja zachodzi wtedy, gdy jedna klasa wykorzystuję drugą klasę, ale nie są od siebie zależne. Agregacja zachodzi wtedy, gdy klasa zawiera drugą klasę, ale współdzieli odwołanie do niej z inną klasą. Natomiast kompozycja zachodzi wtedy, gdy klasa zawiera drugą klasę i są one od siebie zależne (jedna z klas jest odpowiedzialna za tworzenie elementów drugiej klasy, które grupuje).



RYSUNEK 1. DIAGRAM KLAS

Schematy blokowe

Aplikacja składa się z dwóch widoków. Po uruchomieniu aplikacji wyświetlana jest strona startowa, na której znajdują się zasady gry, interaktywna kontrolka dla danych wejściowych (za pomocą, którego użytkownika wprowadza próg punktowy dla rozgrywki) oraz przycisk uruchomiający grę.

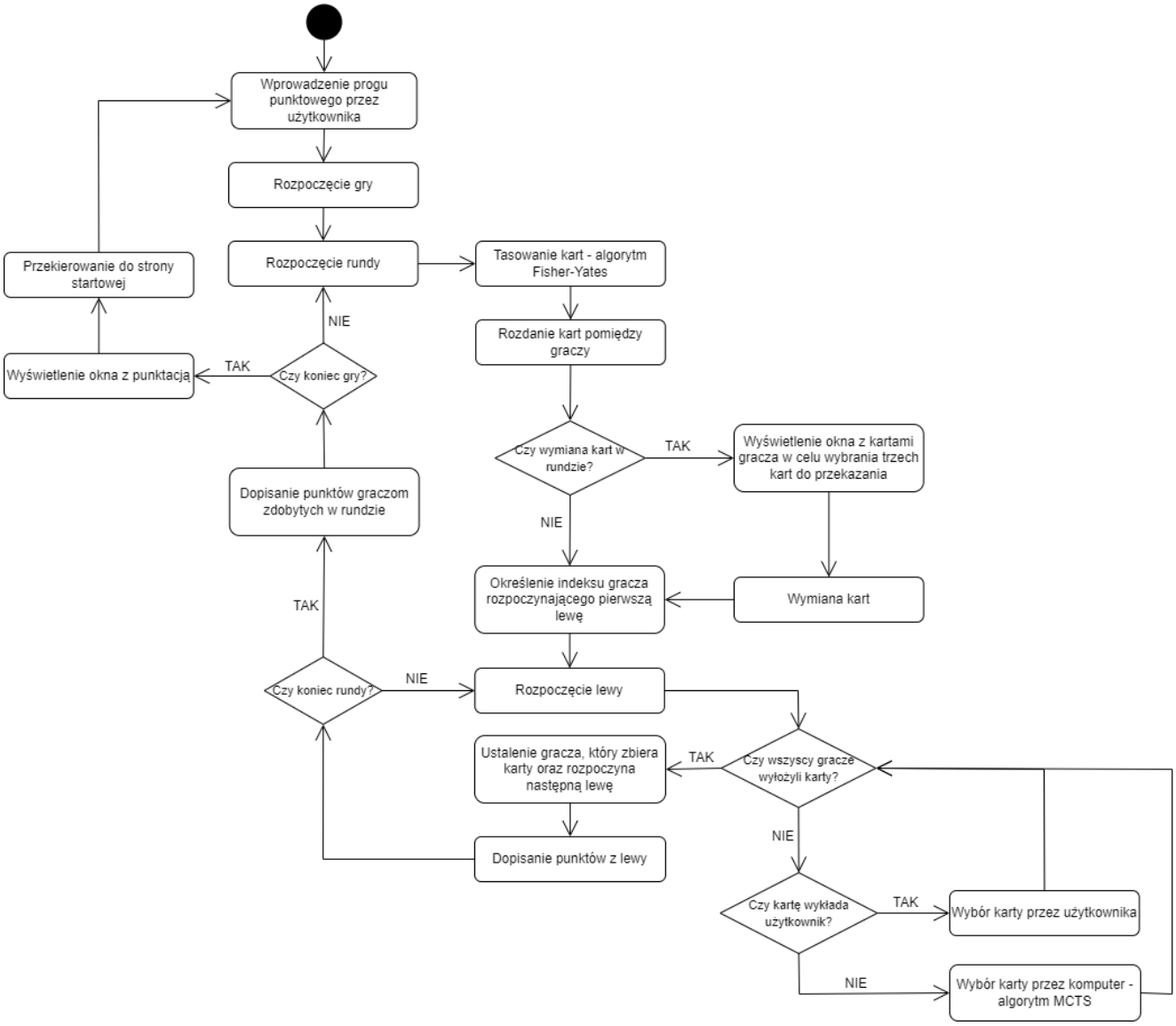
Po uruchomieniu gry następuję wywołanie konstruktora klasy *Game*, do którego przekazany zostaje ustalony przez użytkownika próg punktowy. Domyślna wartość progu punktowego to 100.

W konstruktorze klasy *Game* zostają utworzone obiekty reprezentujące graczy (jeden obiekty typu *Player,* trzy obiekty typu *Computer*). Rozgrywka składa się z rund, a każda runda składa się z 13 lew. Rundy są rozgrywane do momentu kiedy jeden z graczy przekroczy próg punktowy. Wygrywa gracz z najmniejszą ilością punktów.

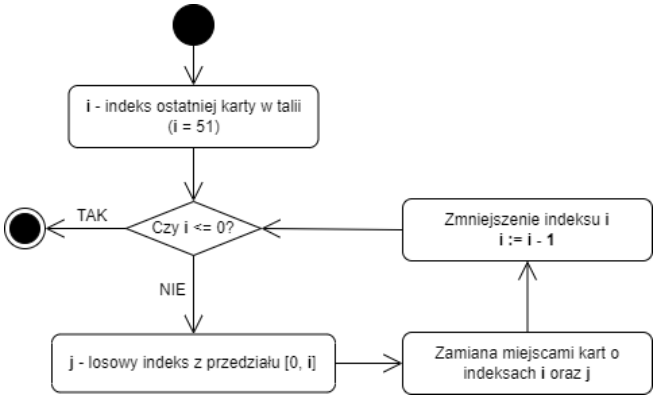
Etapy gry:

1. Tasowanie, rozdanie oraz wymiana kart.
2. Ustalenie indeksu gracza rozpoczynającego lewę (gracz, który posiada dwójkę trefl).
3. Lewa:
   3.1. Wyłożenie kart przez graczy.
   3.2. Ustalenie gracza, który zbiera lewę.
   3.3. Zabranie lewy i ustawienie indeksu użytkownika, który rozpoczyna następną lewę.
   3.4. Dopisanie punktów z lewy.
4. Sprawdzenie czy koniec rundy (gracze nie posiadają już kart), jeśli nie to punkt 3.
5. Dopisanie punktów graczom.
6. Sprawdzenie czy koniec gry, jeśli nie to punkt 1.
7. Wyświetlenie wyników dla danej rozgrywki.

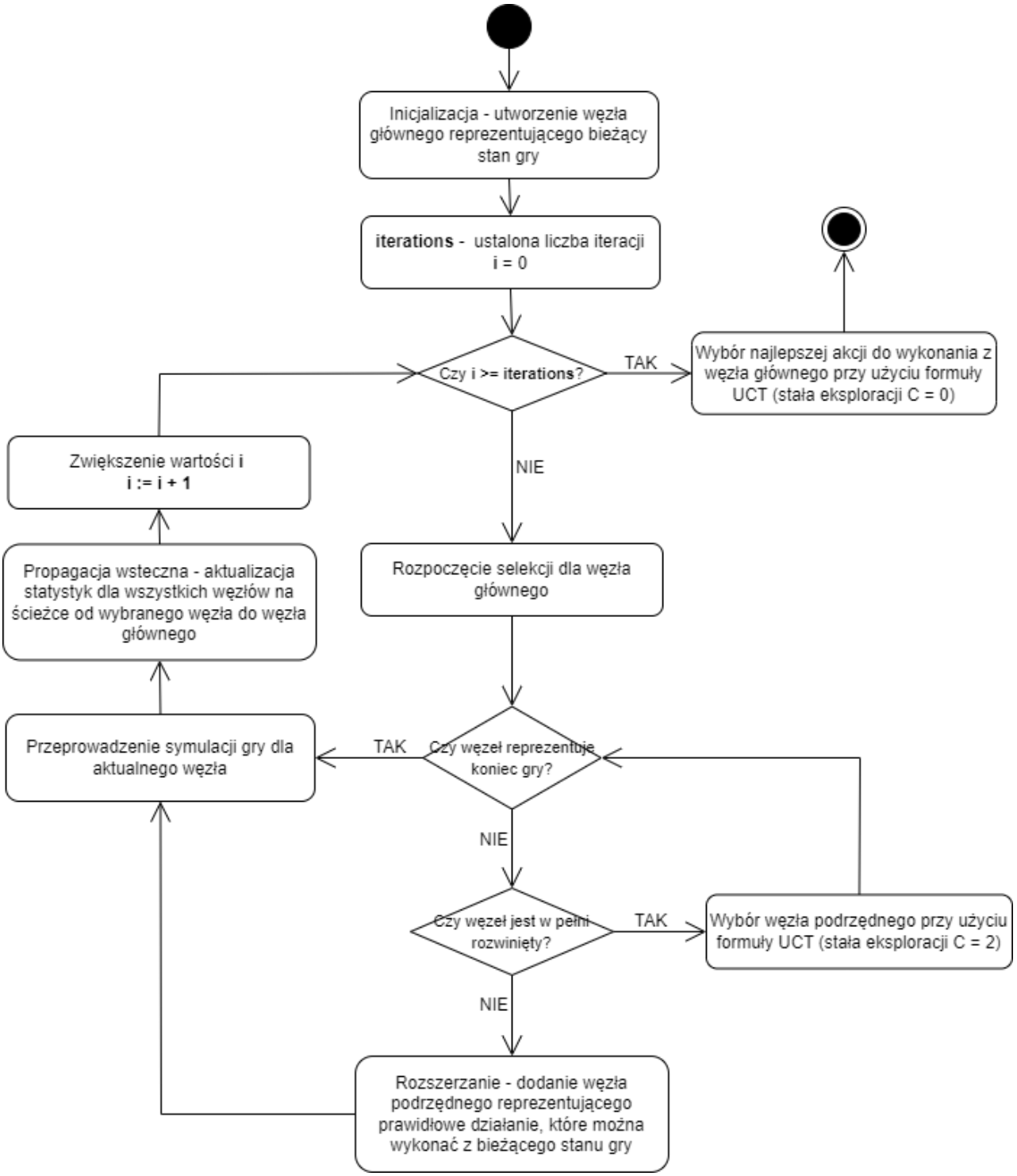
Poniższe diagramy powstał przy użyciu https://app.diagrams.net/



RYSUNEK 2. DIAGRAM CZYNNOŚCI



RYSUNEK 3. DIAGRAM - ALGORYTM FISHER-YATES



RYSUNEK 4. DIAGRAM - ALGORYTM MONTE CARLO TREE SEARCH

Kod źródłowy

Tasowanie kart – algorytm Fisher–Yates

```
shuffle(){
  for(let i = this.numberOfCards - 1; i > 0; i--){
    const j = Math.floor(Math.random() * (i + 1));
    const tmp = this.cards[j];
    this.cards[j] = this.cards[i];
    this.cards[i] = tmp;
  }
}
```

Monte Carlo Tree Search

```
class Node {
  constructor(game, action = null, parent = null){
    this.game = game;
    this.action = action;
    this.isFinal = this.game.isEndOfRound();
    this.isFull = this.isFinal;
    this.parent = parent;
    this.visits = 0;
    this.score = 0;
    this.children = new Map();
  }

  expand(){
    const allowedActions = this.game.getAllowedCards();
    for(let action of allowedActions){
      const key = `${action.value}${action.color}`;

      if(!this.children.has(key)){
        const actionGame = this.game.clone();
        actionGame.simulateMove(action);
        const newNode = new Node(actionGame, action, this);
        this.children.set(key, newNode);

        if(allowedActions.length === this.children.size) this.isFull = true;
        return newNode;
      }
    }
  }
}
```

```
class MCTS {
  constructor(playerIndex){
    this.root = null;
    this.playerIndex = playerIndex;
  }

  search(game, iterations){
    this.root = new Node(game);

    for(let i = 0; i < iterations; i++){
      const node = this.select(this.root);
      const score = this.simulate(node.game);
      this.backpropagate(node, score);
    }

    return this.getBestMove(this.root, 0);
  }
```

```
select(node){
  while(!node.isFinal){
    if(node.isFull){
      const bestMove = this.getBestMove(node, 2);
      node = bestMove.node;
    }else{
      return node.expand();
    }
  }

  return node;
}
```

```
simulate(game){
  let gameCopy = game.clone();
  while(!gameCopy.isEndOfRound()){
    const allowedActions = gameCopy.getAllowedCards();
    const action = gameCopy.players[gameCopy.playerIndex].selectCard(
      allowedActions,
      gameCopy.allCards,
      gameCopy.trick.cardsOnTable,
      gameCopy.trickColor
    );

    gameCopy.simulateMove(action);
  }

  const points = gameCopy.checkRound();
  return 26 - points[this.playerIndex];
}
```

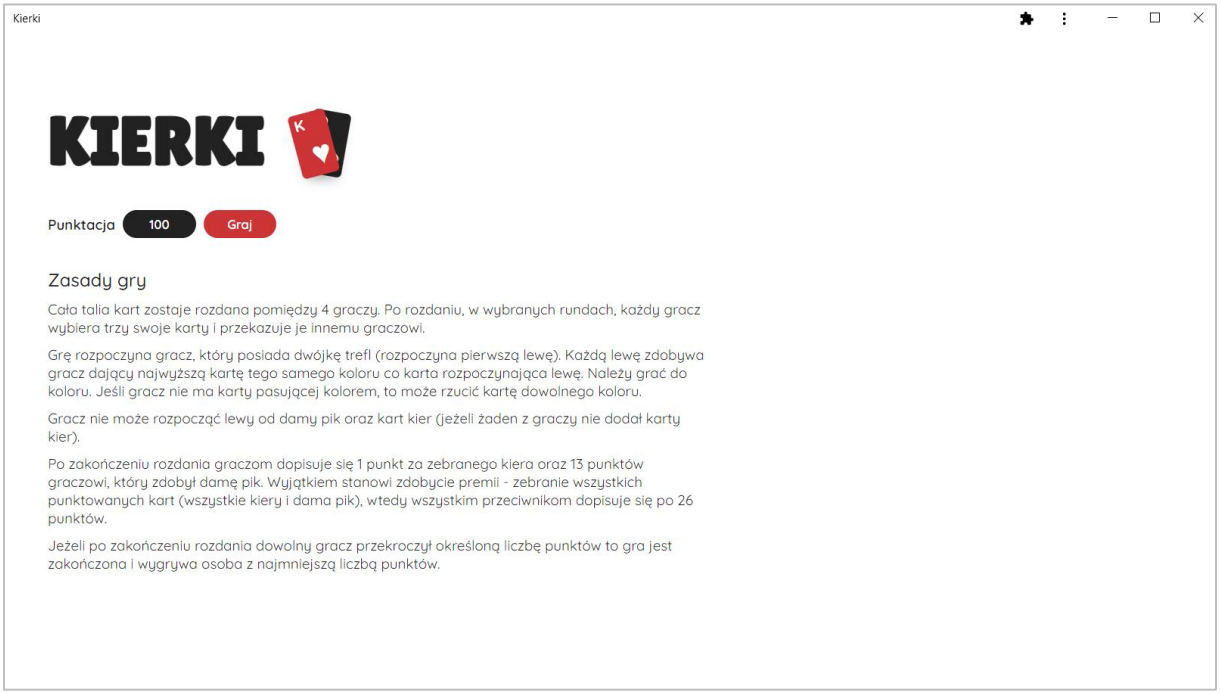
```
backpropagate(node, score){
  while(node){
    node.visits += 1;
    node.score += score;
    node = node.parent;
  }
}
```

```
getBestMove(node, explorationConstant){
  let bestScore = -Infinity;
  let bestNodes = [];

  node.children.forEach(child => {
    const score = child.score / child.visits + explorationConstant * Math.sqrt(Math.log(node.visits) / child.visits);
    if(score > bestScore){
      bestScore = score;
      bestNodes = [child];
    }else if(score === bestScore){
      bestNodes.push(child);
    }
  });

  const randomIndex = Math.floor(Math.random() * bestNodes.length);

  return {
    action: bestNodes[randomIndex].action,
    node: bestNodes[randomIndex]
  };
}
```

Interfejs

Aplikacja zapewnia intuicyjny i łatwy w obsłudze interfejs użytkownika. Treść aplikacji jest skalowana odpowiednio do rozdzielczości ekranu. W projekcie zastosowano RWD (Responsive Web Design).

Po uruchomieniu aplikacji wyświetlana jest strona startowa, na której znajdują się zasady gry, interaktywna kontrolka dla danych wejściowych (za pomocą, którego użytkownika wprowadza próg punktowy dla rozgrywki) oraz przycisk uruchomiający grę.



Po rozpoczęciu gry następuje zmiana interfejsu. Jeśli w danej rundzie następuje wymiana kart pomiędzy graczami, to zostaje wyświetlone okno modalne z kartami użytkownika.

Poniżej przedstawione zostały okna modalne. W pierwszym została wybrana jedna karta, przycisk przekazujący karty do gry jest zablokowany do momentu wybrania trzech kart przez użytkownika.

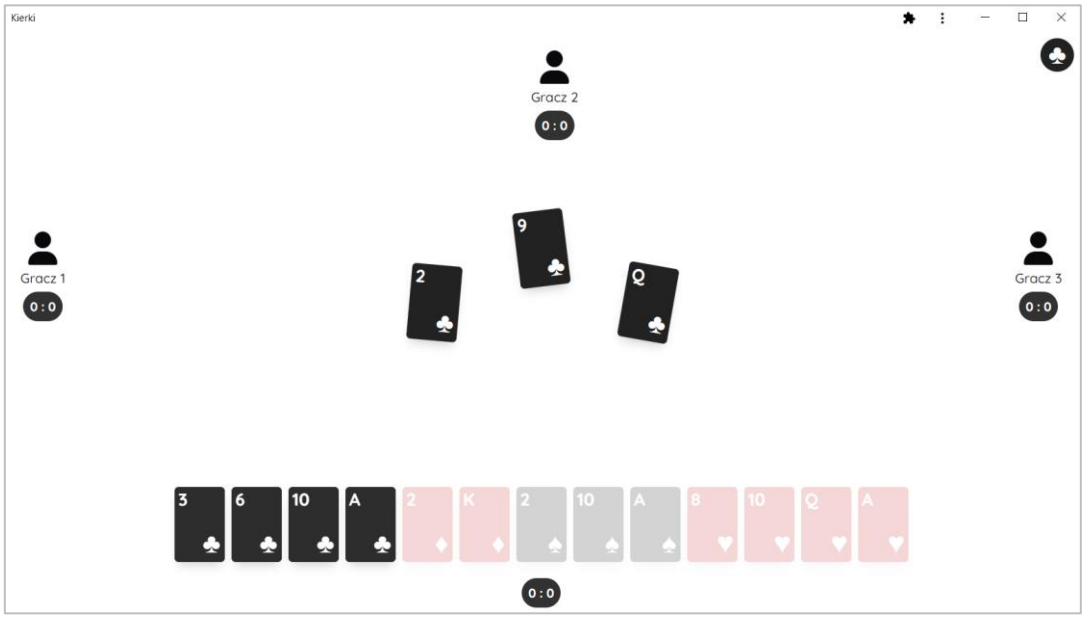
W drugim przypadku zostały wybrane trzy karty, dlatego pozostałe karty zostały zablokowane.

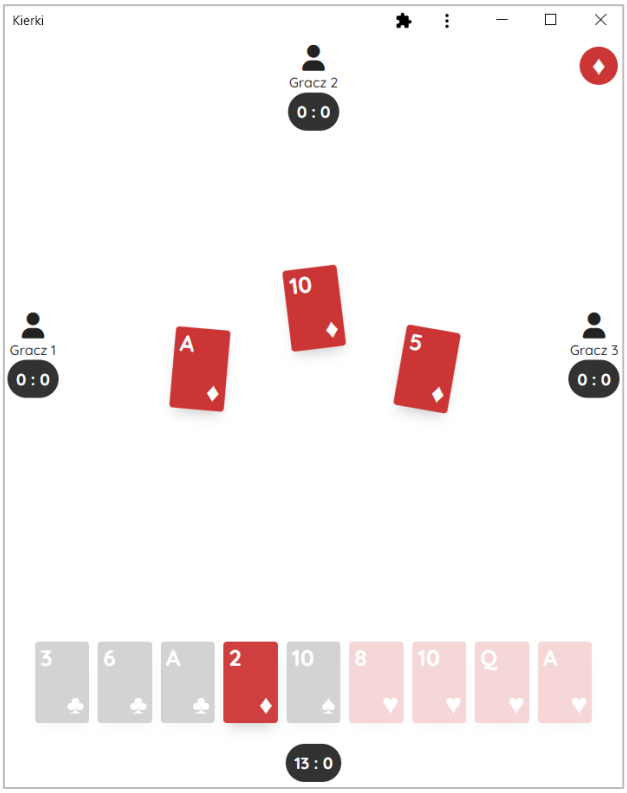
Po przekazaniu kart wyświetlany jest główny interfejs gry, na który składają się:

- Trzech graczy (ikona, nazwa gracza, punktacja)
- Karty użytkownika wraz z punktacją użytkownika,
- Kolor lewy (w prawym górnym rogu),

Format zapisu punktacji „0 : 0” określa ilość punktów zdobytych w aktualnie rozgrywającej rundzie (liczba przed dwukropkiem) oraz ilość punktów zdobytych do tej pory we wszystkich rozegranych rundach (liczba po dwukropku).

Karty, które w danej lewie użytkownik nie może wybrać zostają zablokowane. Podświetlone zostają tylko dozwolone karty w danej lewie.





Po zakończeniu gry wyświetlone zostaje okno modalne wraz z punktacją użytkowników oraz przycisk przekierowujący do strony startowej.

Instrukcja obsługi

Aby uruchomić grę lokalnie należy pobrać i zainstalować Node.js (wraz z Node.js zostanie zainstalowanym npm). Projekt został stworzony w środowisku Node.js wersja 14.17.2 oraz npm wersja 6.14.13.

W głównym folderze gry należy w konsoli użyć polecenia ***npm install*** w celu zainstalowania niezbędnych pakietów. Następnie użyć polecenia ***npm run build,*** spowoduje to utworzenie folderu *dist* z plikami wynikowymi. Aby uruchomić grę należy użyć polecenia ***npm run dev,*** w przeglądarce automatycznie uruchomi się gra (***localhost:3000***).

Powyższa instrukcja dotyczy uruchomienia gry lokalnie wraz z edycją kodu (tryb deweloperski).

Aplikacja (strona) z grą jest dostępna pod adresem https://martazaorska.github.io/gra-kierki/.

Na urządzeniach mobilnych po wejściu na powyższy adres URL istnieje możliwość zainstalowania aplikacji na telefon (możliwość gry w trybie offline).

Kod dostępny pod adresem https://github.com/MartaZaorska/gra-kierki (na tej stronie istnieje również możliwość pobrania projektu w formacie zip).

Bibliografia

Fisher–Yates shuffle. (2023, May 15). Pobrano z lokalizacji Wikipedia, The Free Encyclopedia: https://en.wikipedia.org/wiki/Fisher%E2%80%93Yates_shuffle

Kierki. (2021, Maj 3). Pobrano z lokalizacji Wikipedia: wolna encyklopedia: https://pl.wikipedia.org/wiki/Kierki

Monte-Carlo Tree Search. (2023, Kwiecień 30). Pobrano z lokalizacji Wikipedia: wolna encyklopedia: https://pl.wikipedia.org/wiki/Monte-Carlo_Tree_Search