Poznań, 30.01.2016

Klaudia Anioł
Julian Kwieciński
Emilia Marczyńska
Identyfikator zespołu: Z1

Sprawozdanie z laboratorium sztucznej inteligencji
Informatyka niestacjonarna, sem. VII

I. Opis zadania

Skonstruować symulator gry logicznej Pretwa. Powinien posiadać graficzny interfejs użytkownika, powinna istnieć możliwość rozegrania gry z graczem komputerowym oraz opcjonalna gra dla dwóch graczy.

Zasady gry:

1. Plansza składa się z 19 pól - jedno pole centralne oraz 3 koncentryczne okręgi zawierające po 6 pól.
2. Każdy z graczy posiada po 9 pionów w odmiennym kolorze (użyto koloru czerwonego i czarnego).
3. W początkowym ustawieniu każdy z graczy układa 9 pionów na własnej połowie planszy, wzdłuż 9 promieni, tak aby centralne pole pozostało wolne.
4. W trakcie rozgrywki gracze wykonują ruchy na przemian. W ramach ruchu gracz może przesunąć pion na sąsiednie, połączone linią, wolne pole lub dokonać bicia. Bicie polega na przeskoczeniu pionu przeciwnika na znajdujące się za nim wolne pole - po okręgu lub wzdłuż promienia koła na zewnątrz lub do wewnątrz. Można dokonać wielokrotnego bicia w jednym ruchu, zmieniając w trakcie kierunek ruchu. Zbity pion zostaje usunięty z planszy, a bicie jest obowiązkowe.
5. Zwycięzcą zostaje gracz, który zredukował ilość pionów przeciwnika do 3 lub mniejszej, lub spowodował, że ten nie może wykonać żadnego ruchu, ponieważ jego piony zostały zablokowane (sytuacja patowa).
6. W sytuacji, kiedy żaden z graczy nie może wykonać kolejnego ruchu, wygrywa ten z większą ilością pionów na planszy. Jeśli obaj gracze mają tę samą ilość pionów na planszy, gra kończy się remisem.

II. Założenia realizacyjne

1. Założenia dodatkowe

Gra została zrealizowana zgodnie z opisem umieszczonym w punkcie I. Nie wprowadzono dodatkowych założeń.

2. Algorytmy używane do rozwiązania zadań

a) Algorytm wyszukiwania najlepszego ruchu w drzewie gry

Dane: *boardState* - plansza gry z pozycjami pionków; *nextMove* - kolor gracza, który jest aktualnie na posunięciu.
Wyjście: *(moveFrom, moveTo)*- dane opisujące wybrany przez funkcję ruch na planszy.
Metoda:

1. Wygeneruj ciąg wszystkich możliwych ruchów do wykonania na podstawie planszy gry.
2. Dla każdego ruchu z ciągu wywołaj następujące operacje:
    a. Wygeneruj ciąg możliwych ruchów na podstawie zadanej sytuacji.
    b. Oceń każdy z możliwych ruchów wywołując dla niego algorytm oceny stanu planszy rozpoczynając od poziomu = 3:
        i. jeśli poziom = 0 lub jeden z graczy jest na pozycji przegranej, zwróć ocenę stanu planszy,
        ii. w innym wypadku dodaj do ciągu listOfValues wartość zwróconą przez wywołany rekurencyjnie algorytm oceny stanu planszy na każdym elemencie ciągu ruchów,
        iii. jeśli na posunięciu jest komputer, zwróć największa z wartości z ciągu listOfValues,
        iv. w przeciwnym razie zwróć wartość najmniejszą.
    c. dodaj do ciągu listMoves pary danych: [(ruch, składający się z pól początkowego, pośredniego i końcowego), (ocena ruchu)].
3. Wybierz z ciągu listMoves element o najwyższej ocenie ruchu.
4. Zwróć współrzędne pola źródłowego i docelowego.

b) Algorytm oceny stanu planszy

Dane: *boardState* **-** plansza gry z zapisanymi pozycjami pionków; *color* - kolor gracza, który wykonuje następny ruch i dla którego oceniamy stan planszy (nazywanego dalej Graczem, dla odróżnienia do drugiego gracza, który będzie w algorytmie określany jako Przeciwnik).
Wyjście: wynik obliczeń - numeryczna ocena stanu planszy
Metoda: niech wynik bazowy będzie równy 10

1. Zlicz wszystkie pionki Gracza i przemnóż przez 2.
2. Zlicz wszystkie pionki Przeciwnika i przemnóż przez 2.
3. Zapisz różnicę pomiędzy wartością obliczoną dla Gracza a wartością obliczoną dla Przeciwnika jako value1.
4. Zlicz wszystkie możliwe bicia dla Gracza i przemnóż przez 5.
5. Zapisz obliczoną wartość jako value2.

6. Sprawdź, czy pozycja dla Gracza jest zwycięska dla Gracza.
    a. jeśli Gracz jest zwycięzcą, przypisz wartość 50 do zmiennej value3, jeśli nie jest, przypisz zmiennej wartość 0.
7. Sprawdź, czy Gracz przegrał pojedynek, jeśli tak, zwróć wynik = 0.
8. W przeciwnym wypadku sprawdź, który z graczy jest na posunięciu:
    a. jeśli gracz czerwony (czyli komputer), zwróć wynik:
    wynik = wynik bazowy + value1 + value2 + value3,
    b. jeśli na posunięciu jest gracz czarny, zwróć wynik:
    wynik = ((wynik bazowy + value1 + value2 + value3) * (-1) + 25).

3. Języki programowania, narzędzia, środowisko implementacji

- Język C#/.NET Framework 4.0 - środowisko graficzne aplikacji,
- Język F# - mechanika gry oraz sztuczna inteligencja,
- Środowisko programistyczne: Microsoft Visual Studio 2015.

III. Podział prac

Autor	Podzadanie
Klaudia Anioł	Projekt aplikacji i jej działania, współudział w wyborze i projekcie najważniejszych algorytmów i funkcji. Stworzenie interfejsu graficznego aplikacji - wykonanie planszy i menu gry w języku C#. Testowanie działania aplikacji, współudział w tworzeniu sprawozdania końcowego.
Julian Kwieciński	Projekt aplikacji i jej działania, współudział w wyborze i projekcie najważniejszych algorytmów i funkcji. Implementacja części opisującej zasady i mechanikę gry, pomoc w tworzeniu interfejsu graficznego. Współudział w tworzeniu sprawozdania końcowego.
Emilia Marczyńska	Projekt aplikacji i jej działania, współudział w wyborze i projekcie najważniejszych algorytmów i funkcji. Implementacja części logicznej gry (głównie algorytm mini-maks oraz funkcja oceniająca stan planszy), testowanie działania aplikacji, współudział w tworzeniu sprawozdania końcowego.

IV. Opis implementacji

1. Struktury danych.

a) Podstawowa struktura danych w aplikacji reprezentuje aktualny stan planszy (BoardState). Jest to kolekcja 19 par klucz - wartość, gdzie kluczem jest pole gry, a przypisaną mu wartością - stan tego pola. Położenie pola jest określanie jako krotka reprezentująca jego współrzędne lub informacja, że jest to pole centralne, natomiast stan poprzez informację, czy na danym polu znajduje się pionek koloru czerwonego, czarnego, czy też pole jest puste.

$$\text{Stan planszy} = [(P_1, S_1), (P_2, S_2), \ldots, (P_{19}, S_{19})],$$

gdzie:

- P_i - pole na planszy określone przez współrzędne,
- S_i - stan danego pola.

b) nextMove - zmienna przechowuje informację o tym, jaki powinien odbyć się kolejny ruch. Zmienna może być dwojakiego typu:

- informacja o tym, jaki kolor pionka będzie wykonywał kolejny ruch,
- współrzędne aktualnego pola, jeśli bicie z tego pola jest możliwe do wykonania.

c) struktura Move reprezentująca ruch pionka na planszy zawiera elementy:

- początkowy stan planszy
- aktualne współrzędne pola,
- ciąg kolejnych ruchów pionka (zakładamy, ze przy biciu może być wykonanych kilka ruchów z rzędu).

2. Najważniejsze funkcje i procedury zdefiniowane w programie.

Poniżej opisano najważniejsze funkcje wykorzystywane w programie. Każda funkcja jest określona za pomocą jej nazwy, listy argumentów, które przyjmuje (umieszczone w nawiasach) oraz argumentów zwracanych, które umieszczono po dwukropku.

2.1. defaultBoardState(): allFields.

- allFields - zbiór par dla wszystkich pól planszy (współrzędne pola, stan pola).

Funkcja bezargumentowa tworzy początkowy stan planszy. Najpierw tworzone jest pole centralne, potem, po wygenerowaniu liczb oznaczających maksymalną wartość okręgów i linii przecinających okręgi na planszy, tworzone są pozostałe pola. Pola po prawej stronie od pola centralnego otrzymują kolor czerwony, a po lewej stronie kolor czarny. Dla każdego pola tworzona jest krotka składająca się ze współrzędnych pola i jego stanu. Wszystkie wygenerowane krotki dodawane są do kolekcji allFields.

2.2. validMovesForColor (color, boardState): walkMoves.

- color - kolor pionka,
- boardState - aktualny stan planszy,
- walkMoves - lista dostępnych ruchów, określona jako zbiór par współrzędnych pól - pola, z którego ma być wykonany ruch oraz pola, na który ma być przesunięty pionek.

Funkcja generuje listę ruchów dostępnych do wykonania dla danego koloru pionka. Sprawdza możliwości posunięć na sąsiednie pola oraz możliwości bić. Jeśli są dostępne bicia, zwracana jest tylko lista możliwych bić.

2.3. applyMove (ffrom, fto, boardState): (nextState, nextMove).

- ffrom, fto - zmienne opisujące położenie pionka przed i po wykonanym ruchu,
- boardState - aktualny stan planszy,
- nextState - stan planszy po ruchu, struktura typu boardState,
- nextMove - zmienna typu nextMove, dotyczy ruchu, który będzie mógł być wykonany po wykonaniu bieżącego ruchu.

Funkcja generuje nowy stan planszy po wykonaniu ruchu. W trakcie sprawdza, jaki kolor pionka znajduje się na polu, z którego ma zostać wykonany ruch, generuje listę dostępnych ruchów i sprawdza, czy zaplanowany do wykonania ruch jest prawidłowy. Jeśli pole, z którego gracz chce wykonać ruch, jest puste, również zwraca informację o nieprawidłowym ruchu.

2.4. shannonFunction (boardState, nextMove, level): value.

- boardState - aktualny stan planszy,
- nextMove - zmienna typu nextMove, dotyczy aktualnie mającego odbyć się ruchu,
- level - poziom przeszukiwania w funkcji,
- value - wartość najlepiej ocenionego ruchu.

Funkcja rekurencyjna przeszukuje drzewo możliwych ruchów dla aktualnej sytuacji i zdefiniowanej głębokości. Po dotarciu do zdefiniowanego poziomu, bądź sytuacji przegranej jednego z graczy, wywoływana jest funkcja oceniająca dla stanu planszy z punktu widzenia gracza znajdującego się na posunięciu. Wartość po ocenie stanu planszy dodawana jest do ciągu zawierającego wszystkie oceny ruchów na końcowym etapie przeszukiwania. Jeśli na posunięciu znajduje się komputer, zwracana jest maksymalna wartość z listy, jeśli przeciwnik, zwracana jest minimalna wartość.

2.5. moveOfTheComputer (boardState, nextMove): (from, to).

- boardState - aktualny stan planszy,
- nextMove - zmienna typu nextMove, dotyczy aktualnie mającego odbyć się ruchu,
- from- współrzędne pionka wykonującego ruch,
- to - współrzędne docelowe wykonywanego ruchu.

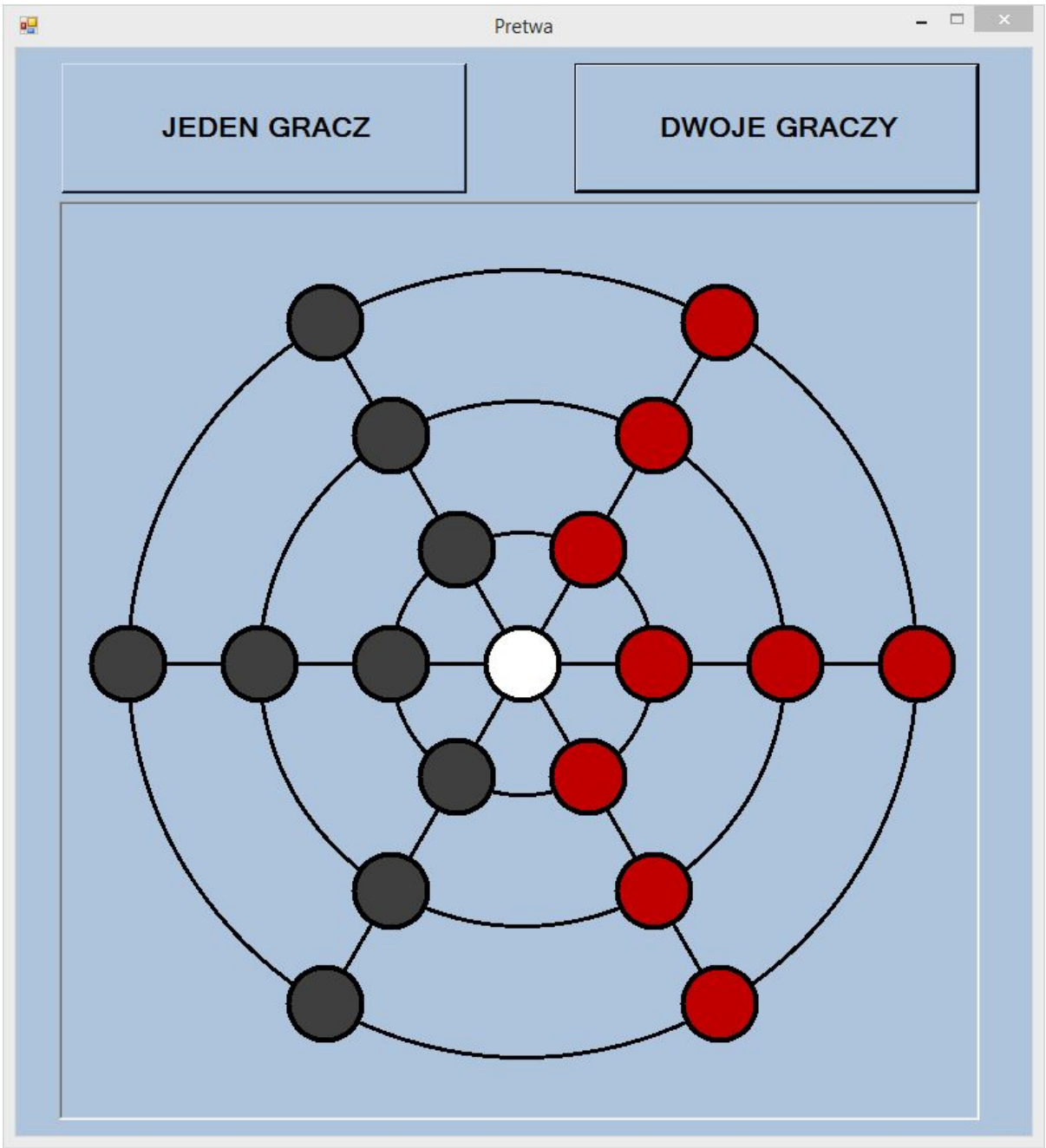
Funkcja najpierw generuje ciąg wszystkich możliwych do wykonania ruchów dla gracza, którym jest komputer. Następnie dla każdego ruchu z listy wywołuje funkcję Shannona (opisana powyżej) dla zdefiniowanej głębokości przeszukiwania. Głębokość zdefiniowana w funkcji = 3. Tworzony jest ciąg par (ruch, ocena ruchu). Zwracany jest ruch oceniony najwyżej.

V. Użytkowanie i testowanie systemu

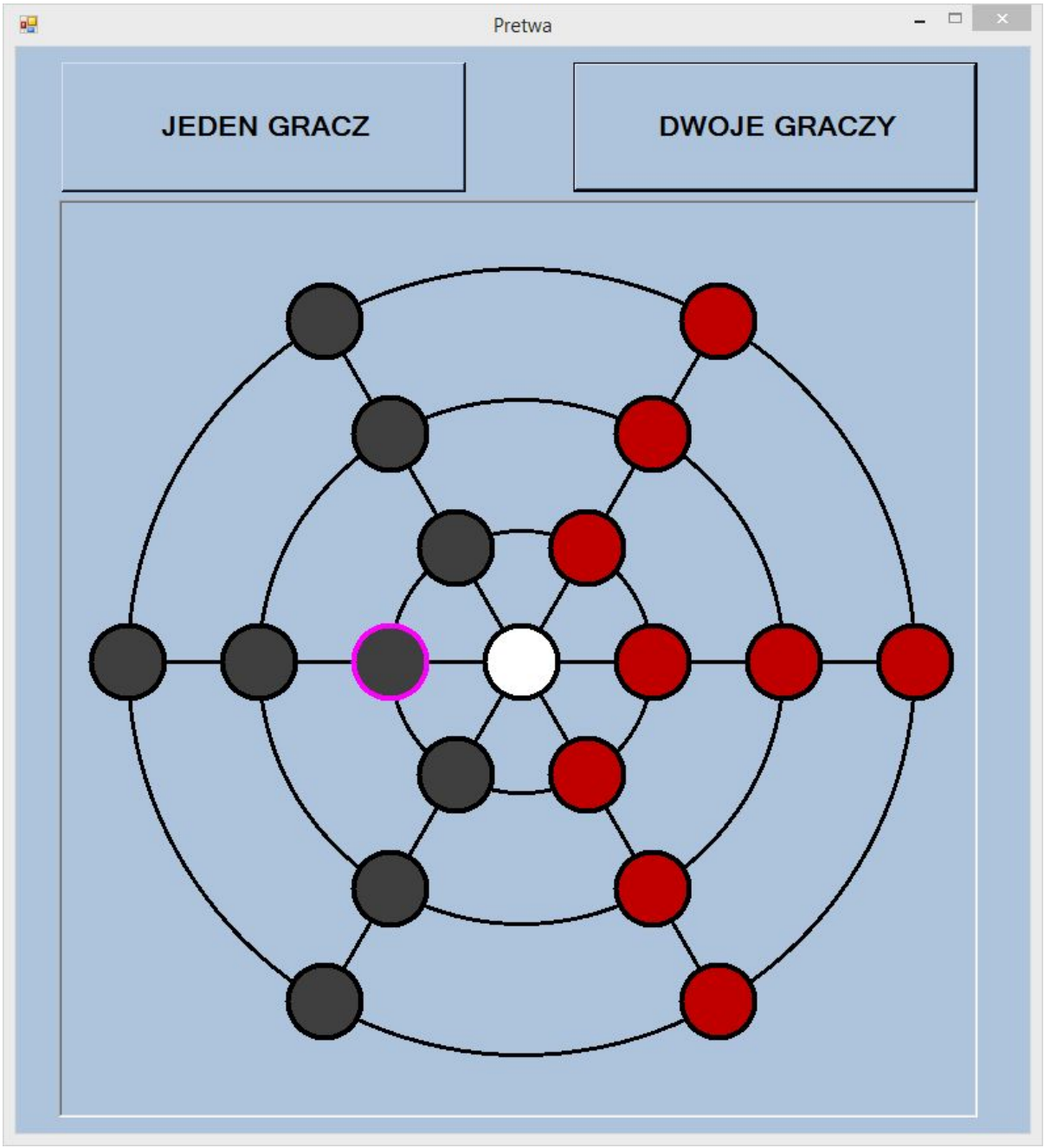
Opis planszy:

- pionki czerwone - komputer
- pionki czarne - gracz

Sterowanie grą odbywa się za pomocą myszy. Po umieszczeniu kursora myszy nad pionkiem, jeżeli może on wykonać w danej sytuacji ruch, pionek jest podświetlany.
Po uruchomieniu gry wyświetla się plansza z pionkami rozstawionymi odpowiednio czarne z lewej strony oraz czerwone z prawej. Na środku pozostaje puste pole centralne.



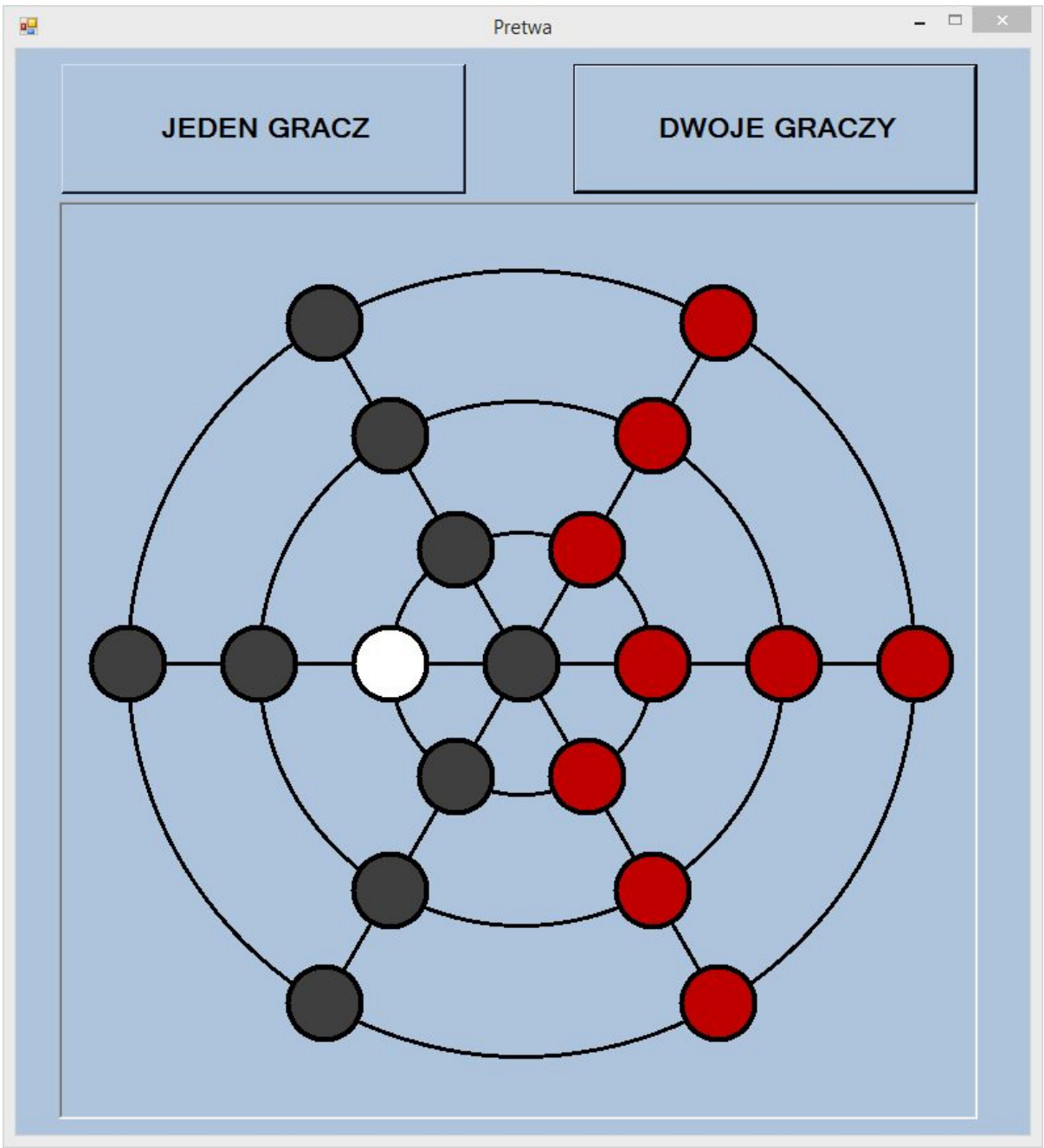
Kliknięcie pionka który może w danej sytuacji wykonać ruch powoduje jego zaznaczenie.



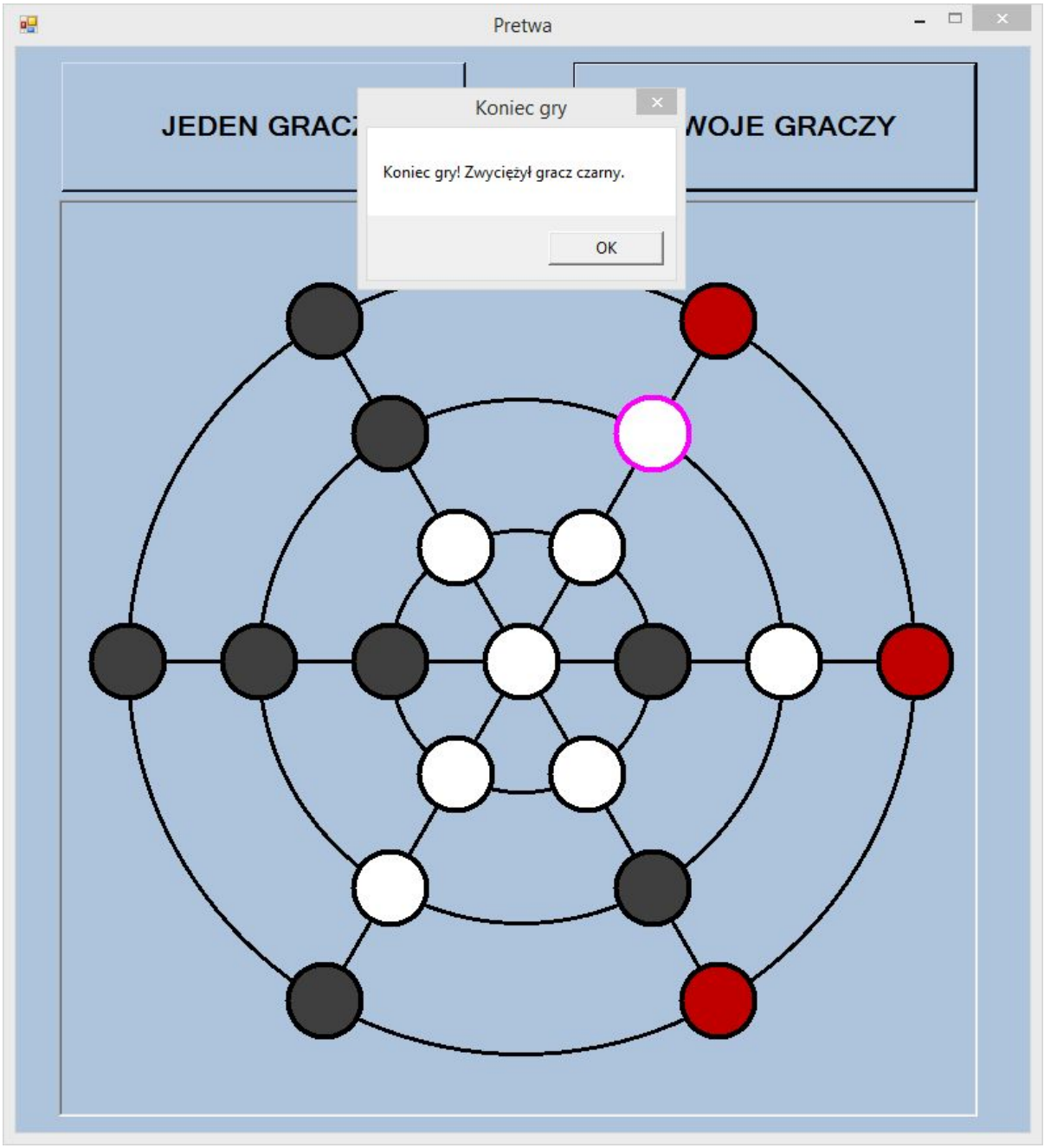
Jeżeli pionek jest zaznaczony, możliwe są dwie dalsze czynności:

- zaznaczenie innego pionka który może wykonać ruch
- wybranie pustego pola na które dany pionek może wykonać ruch lub bicie

Po wybraniu pola docelowego wykonywany jest ruch:



Gra kończy się gdy jeden z graczy przegra. Gracz uznawany jest za przegranego gdy w grze pozostaną mniej niż cztery pionki jego koloru lub gdy nie będzie mógł wykonać żadnego ruchu (zostanie zablokowany).



Kliknięcie w dowolnym momencie przycisku “Jeden gracz” lub “Wielu graczy” powoduje zakończenie trwającej rozgrywki i ponowne ułożenie pionków na planszy.

VI. Kod programu

1. Types.fs - wykorzystywane struktury danych

```
namespace Pretwa

type public Player =
      | Red
      | Black

type public FieldState =
      | Color of Player
      | Empty

type public FieldCoords =
      | Edge of int*int
      | Center

type public NextMove =
      | Color of Player
      | Piece of FieldCoords

type public BoardState = Map<FieldCoords, FieldState>

type public Move = {
      State : BoardState
      Start : FieldCoords
      Path : List<FieldCoords>
}
```

2. Utils.fs - funkcje pomocnicze

```
namespace Pretwa
module Utils =
	let oneof list item = List.exists(fun x -> x = item) list

	let cartesian set1 set2 =
	[for x in set1 do
		for y in set2 do
		yield (x,y)]

	module Circle =
	let put value (cMin, cMax) =
		let offset = cMin
		let tmpMax = cMax - offset
		let count = cMax - cMin + 1
		let tmpValue = value - offset
		offset +
			if tmpValue < 0
			then tmpMax - abs(tmpValue % count) + 1
			else tmpValue % count

	let up value (cMin, cMax) = put (value + 1) (cMin, cMax)

	let down value (cMin, cMax) = put (value - 1) (cMin, cMax)
```

```
let distance val1 val2 (cMin, cMax) =
       let p1 = put val1 (cMin, cMax)
       let p2 = put val2 (cMin, cMax)
       let dist = abs(p1-p2)
       let count = abs(cMin-cMax) + 1
       min(dist, count - dist)

let mapMerge group1 group2 appender =
      group1 |> Seq.fold(fun (acc:Map<'a,'b>) (KeyValue(key, values)) ->
            match acc.TryFind key with
            | Some items -> Map.add key (appender values items) acc
            | None -> Map.add key values acc) group2

let joinMaps map1 map2 = mapMerge map1 map2 Seq.append
```

3. Board.fs - mechanika gry: obsługa stanu planszy, weryfikacja poprawności wykonywanych ruchów, generowanie listy ruchów dostępnych dla danego stanu planszy, algorytm obliczający najbardziej korzystny ruch.

```
namespace Pretwa
open Pretwa.Utils
open System

module Board =
	let edgeCount = 3
	let edgeLength = 6
	let maxEdge = edgeCount - 1
	let maxField = edgeLength - 1

	let oppositeField fn = Circle.put (fn + edgeLength / 2) (0, maxField)
	let otherPlayer player =
	match player with
	| Player.Black -> Player.Red
	| Player.Red -> Player.Black

	let makeField (en, fn) =
	if en = -1 then FieldCoords.Center
	elif en < 0 then FieldCoords.Edge((abs en) - 2, oppositeField (Circle.put fn (0, maxField)))
	else FieldCoords.Edge(en, Circle.put fn (0, maxField))

	let walkUp (en, fn) = makeField (en - 1, fn)
	let walkDown (en, fn) = makeField (en + 1, fn)
	let walkLeft (en, fn) = makeField (en, fn - 1)
	let walkRight (en, fn) = makeField (en, fn + 1)

	let jumpUp (en, fn) = (makeField(en - 1, fn), makeField(en - 2, fn))
	let jumpDown (en, fn) = (makeField(en + 1, fn), makeField(en + 2, fn))
	let jumpLeft (en, fn) = (makeField(en, fn - 1), makeField(en, fn - 2))
```

```
let jumpRight (en, fn) = (makeField(en, fn + 1), makeField(en, fn + 2))

let insideBoard field =
match field with
| FieldCoords.Center -> true
| FieldCoords.Edge(en,ef) -> en >= 0 && en < edgeCount && ef >= 0 && ef < edgeLength

let outOfBoard field = insideBoard field |> not

let fieldState field (boardState: BoardState) =
boardState.Item field

let adjacentWalkFields field =
if outOfBoard field then failwith "Out of board!"
match field with
| FieldCoords.Center -> [0..maxField] |> List.map (fun x -> FieldCoords.Edge(0,x))
| FieldCoords.Edge(en, ef) ->
       [walkUp (en, ef)] @
       [walkDown (en, ef)] @
       [walkLeft (en, ef)] @
       [walkRight (en, ef)]
       |> List.where insideBoard

let adjacentJumpFields field =
if outOfBoard field then failwith "Out of board!"
match field with
| FieldCoords.Center -> [0..maxField] |> List.map (fun x -> (FieldCoords.Edge(0,x), FieldCoords.Edge(1,x)))
| FieldCoords.Edge(en, ef) ->
       [jumpUp (en, ef)] @
       [jumpDown (en, ef)] @
       [jumpLeft (en, ef)] @
       [jumpRight (en, ef)]
       |> List.where (fun (f1, f2) -> insideBoard f1 && insideBoard f2)
```

```
let allPieces color boardState =
boardState
|> Map.toList
|> List.where (fun (_, state) -> state = color)
|> List.map (fun (coords, _) -> coords)

let validWalkMovesForField ffrom boardState =
if outOfBoard ffrom then failwith "Out of board!"
adjacentWalkFields ffrom
|> List.where (fun field -> fieldState field boardState = FieldState.Empty)

let validJumpMovesForField ffrom  boardState =
if outOfBoard ffrom then failwith "Out of board!"
match fieldState ffrom boardState with
| FieldState.Empty -> []
| FieldState.Color c0 ->
       adjacentJumpFields ffrom
       |> List.where (fun (f1, f2) ->
       fieldState f2 boardState = FieldState.Empty &&
       match fieldState f1 boardState with
       | FieldState.Empty -> false
       | FieldState.Color c1 -> c0 <> c1)

let validMovesForField ffrom boardState =
match validJumpMovesForField ffrom boardState with
| [] -> validWalkMovesForField ffrom boardState |> List.map (fun fto -> (ffrom, None, fto))
| list -> list |> List.map(fun (fjump, fto) -> (ffrom, Some fjump, fto))

let validJumpMovesForColor color boardState =
allPieces color boardState
|> List.map (fun field -> (validJumpMovesForField field boardState) |> List.map(fun (mid, dst) -> (field, mid, dst)))
|> List.concat
```

```
let validWalkMovesForColor color boardState =
let pieces = allPieces color boardState
let jumpsAvailable =
       allPieces color boardState
       |> List.exists (fun field ->
       match validJumpMovesForField field boardState with
       | [] -> false
       | _ -> true)
if jumpsAvailable
       then []
       else
       pieces
       |> List.map (fun field -> (validWalkMovesForField field boardState) |> List.map (fun dst -> (field, dst)))
       |> List.concat

let validMovesForColor color boardState =
let pieces = allPieces color boardState
let walkMoves = validWalkMovesForColor color boardState
match walkMoves with
| [] ->
       validJumpMovesForColor color boardState
       |> List.map (fun (src, mid, dst) -> (src, Some mid, dst))
| _ ->
       walkMoves
       |> List.map(fun (src, dst) -> (src, None, dst))

(*let allValidMoves boardState =
(validMovesForColor (FieldState.Color(Player.Black)) boardState) @
(validMovesForColor (FieldState.Color(Player.Red)) boardState)*)

let isColor nextMove =
match nextMove with
```

```
| NextMove.Color(Player.Red) -> true
| NextMove.Color(Player.Black) -> true
| NextMove.Piece(_) -> false

let isBlack nextMove =
match nextMove with
| NextMove.Color(Player.Red) -> false
| NextMove.Color(Player.Black) -> true
| NextMove.Piece(_) -> false

let getField (Piece p) = p

let allValidMoves nextMove boardState =
let validmoves  =
       if isColor(nextMove) && isBlack(nextMove)
       then (validMovesForColor (FieldState.Color(Player.Black)) boardState)
       elif isColor(nextMove)
       then (validMovesForColor (FieldState.Color(Player.Red)) boardState)
       else (validMovesForField (getField nextMove) boardState)
(validmoves)



let isValidMove ffrom fto boardState =
validMovesForField ffrom boardState
|> List.exists (fun (f1, _, f3) -> f1 = ffrom && f3 = fto)

let applyMove ffrom fto boardState =
let color =
       match fieldState ffrom boardState with
       | FieldState.Color c -> c
       | Empty -> failwith "Invalid move!"
match validMovesForField ffrom boardState |> List.where (fun (f1, _, f2) -> f2 = fto) with
| [(f1, None, f2)] ->
```

```
        let nextState =
        boardState
        |> Map.add ffrom FieldState.Empty
        |> Map.add fto (FieldState.Color color)
        //|> (fun bs -> (bs, (NextMove.Color color)))
        let nextMove = NextMove.Color( otherPlayer color)
        (nextState, nextMove)
| [(f1, Some f2, f3)] ->
        let nextState =
        boardState
        |> Map.add f1 FieldState.Empty
        |> Map.add f2 FieldState.Empty
        |> Map.add f3 (FieldState.Color color)
        let nextMove = if validJumpMovesForField f3 nextState <> [] then NextMove.Piece(f3) else NextMove.Color( otherPlayer color)
        (nextState, nextMove)
| _ -> failwith "Invalid move!"

let getFirstFromTuple tuple1 =
match tuple1 with
| (a, b, c) ->
        let first = a
        (first)

let getThirdElementFromTuple tuple2 =
match tuple2 with
| (a, b, c) ->
        let third = c
        (third)

let hasPlayerLost color boardState =
let pieces = allPieces color boardState
if pieces.Length <= 3
then true
```

```
else (validMovesForColor color boardState).Length = 0

let getPlayer =
let nextmove :NextMove = NextMove.Color(Player.Black)
(nextmove)

let getColor =
let fieldstate = FieldState.Color(Player.Black)
(fieldstate)

let otherColor color =
match color with
| FieldState.Color(Player.Black) -> FieldState.Color(Player.Red)
| FieldState.Color(Player.Red) -> FieldState.Color(Player.Black)

let colorInNextMove nextMove (boardState:BoardState) =
match nextMove with
| NextMove.Color(Player.Red) -> Player.Red
| NextMove.Color(Player.Black) -> Player.Black
| NextMove.Piece(nextMove) ->
       let fs = boardState.[nextMove]
       match fs with
       | FieldState.Color(fs) -> fs

let estimateFunction color boardState =
let baseValue = 10
let myPieces = (allPieces color boardState).Length * 2
let oppositePieces = (allPieces (otherColor color) boardState).Length * 2
let value1 = myPieces - oppositePieces
let jumpsAvailable = validJumpMovesForColor color boardState
let value2 = jumpsAvailable.Length * 5
let oppositeColor = otherColor color
let winning = hasPlayerLost oppositeColor boardState
```

```
let value3 = if winning then 50 else 0

let losing = hasPlayerLost color boardState
let estimatedValue =
        if losing
        then 0
        elif color = FieldState.Color(Player.Red) then value1 + value2 + value3 + baseValue
        else ((value1 + value2 + value3 + baseValue) * (-1) + 25)
(estimatedValue)

let getMovesList nextMove color boardState=
match nextMove with
        | NextMove.Color(_) ->
        (validMovesForColor color boardState)
        | NextMove.Piece(_) ->
        let fieldcoord = getField nextMove
        (validMovesForField fieldcoord boardState)

let getNextSituation move boardState =
let ffrom = getFirstFromTuple move
let tto = getThirdElementFromTuple move
let nextSituation = applyMove ffrom tto boardState
(nextSituation)

let rec shannonFunction boardState nextMove level =
let color = colorInNextMove nextMove boardState
if level = 0 || hasPlayerLost (FieldState.Color(Player.Black)) boardState || hasPlayerLost (FieldState.Color(Player.Red)) boardState
then (estimateFunction (FieldState.Color(color)) boardState)
else
        let moves = getMovesList nextMove (FieldState.Color(color)) boardState
        let listOfValues = [for move in moves do
                        let nextSit = getNextSituation move boardState
                        yield (shannonFunction (fst(nextSit)) (snd(nextSit)) (level-1))]
```

```
        if color = Player.Red
        then
        (List.max listOfValues)
        else
        (List.min listOfValues)


let moveOfTheComputer boardState nextMove =
let movesComp = getMovesList nextMove (FieldState.Color(Player.Red)) boardState
let level = 3
let listMoves = [for move in movesComp do
                     let nextSit = getNextSituation move boardState
                     let value = shannonFunction (fst(nextSit)) (snd(nextSit)) level
                     yield (move, value)]
let maxX = List.maxBy snd listMoves
printfn "Lista listMmoves: %A" listMoves
printfn "MAX: %A" maxX
printfn ""
let moveComp = fst maxX
let moveFrom = getFirstFromTuple moveComp
let moveTo = getThirdElementFromTuple moveComp
(applyMove moveFrom moveTo boardState)
```

```
let defaultBoardState =
        let centerField = (FieldCoords.Center, FieldState.Empty)
        let edgeNumbers = [0..maxEdge]
        let edgeFieldNumbers = [0..maxField]
        let fieldCoordinates = cartesian edgeNumbers edgeFieldNumbers
        let edgeFields =
                fieldCoordinates
                |> List.map (fun (en, ef) ->
                        if ef < edgeLength / 2
                        then (FieldCoords.Edge(en,ef), FieldState.Color(Player.Black))
                        else (FieldCoords.Edge(en,ef), FieldState.Color(Player.Red)))
        let allFields = centerField :: edgeFields
        Map.ofList allFields
```