Projekt ZZSI

Rok akademicki	Rodzaj studiów*: SSI/NSI/NSM	Grupa	Prowadzący
2012/2013	**NSM**	**INF1**	Dr inż. **Piotr Czekalski**

Raport końcowy

Temat:

Dylemat Więźnia

Skład sekcji:	1. Neuman Arkadiusz 2. Płaza Maciej

1. Wprowadzenie

1.1 Opis problemu: Dylemat Więźnia

Dylemat więźnia to jeden z najsłynniejszych problemów w teorii gier. Problem ten został wymyślony przez dwóch pracowników RAND Corporation, Melvina Dreshera oraz Merrilla Flooda w 1950 roku. Zasady gry zostały sformalizowane przez Alberta Tuckera w 1992 roku. W klasycznej formie problem jest przedstawiany następująco:

Dwóch podejrzanych zostało zatrzymanych przez policję. Policja nie mając wystarczających dowodów do postawienia zarzutów rozdziela więźniów i przedstawia każdemu z nich tę samą ofertę: jeśli będzie zeznawać przeciwko drugiemu, a drugi będzie milczeć, to zeznający wyjdzie na wolność, a milczący dostanie dziesięcioletni wyrok. Jeśli obaj będą milczeć, obaj odsiedzą 6 miesięcy za inne przewinienia. Jeśli obaj będą zeznawać, obaj dostaną pięcioletnie wyroki. Każdy z nich musi podjąć decyzję niezależnie i żaden nie dowie się czy drugi milczy czy zeznaje aż do momentu wydania wyroku. Jak powinni postąpić?

Dylemat więźnia jest klasycznym przykładem problemu konfliktu i kooperacji. W zależności od decyzji obu graczy każdy z nich otrzymuję wypłatę zgodnie z macierzą wypłat:

		Gracz 2	
	Decyzja	Współpraca	Zdrada
Gracz 1	Współpraca	Reward, Reward	Sucker's Payoff, Temptation
	Zdrada	Temptation, Sucker's Payoff	Punishment, Punishment

W zależności od podjętych decyzji, uczestnicy mogą otrzymać następujące nagrody:

- Gdy gracze współpracują, to otrzymują umiarkowaną nagrodę (Reward = 3).
- Gdy zdradzi jeden z nich, to dostaję najwyższą nagrodę (Temptation = 5), zaś drugi dostaje wynagrodzenie godne naiwniaka (Sucker's Payoff = 0).
- Gdy gracze postanowią zdradzić, to dostają umiarkowaną karę (Punishment = 2).

Problem staje się ciekawszy, gdy grę będzie się powtarzało, dzięki czemu przy podejmowaniu decyzji o współpracy lub zdradzie gracze będą kierować się historią przeszłych zachowań. Jest to tak zwany iterowany dylemat więźnia.

Problem ten tak zainteresował badaczy zajmujących się teorią gier, że w 1984 roku Robert Axelrod zorganizował turniej, którego przeciwnikami były programy komputerowe przeprowadzające potyczki dotyczące dylematu więźnia. Axelrod zauważył, iż faworytem okazała się prosta strategia „wet za wet", w której to w pierwszym kroku współpracuje się, zaś w kolejnych ruchach powtarza się poprzedni wybór przeciwnika (czyli „odpłacaj pięknym za nadobne"). Dzięki tym spostrzeżeniom został sformułowany algorytm, który wykorzystujemy w naszym programie.

2. Opis klas.

2.1 Main.java

Jest to główna klasa aplikacji, w której aplikacja rozpoczyna działanie. Klasa składa się z następujących metod:

void main(String[] args) – jest to główna metoda aplikacji, która jest wywoływana zaraz po uruchomieniu aplikacji. W niej zawarte jest sczytywanie wprowadzanych przez użytkownika danych (liczbę osobników populacji, liczbę generacji, liczbę rund w jednej rozgrywce, prawdopodobieństwo mutacji, prawdopodobieństwo krzyżowania). Po wpisaniu poprawnych (czyli mieszczących się w zakresach) danych wykonywana się metoda **runGame()** z klasy **Game**.

int getIntNumber(final int min, final int max) – metoda służy do sczytywania liczb całkowitych wpisywanych przez użytkownika. Liczba mieszcząca się w zakresie podanym w argumentach jest zwracana.

double getDoubleNumber(final double min, final double max) – podobnie jak powyższa sczytuje wpisywaną przez użytkownika liczbę zmiennoprzecinkową. Liczba mieszcząca się w zakresie podanym w argumentach jest zwracana.

String getAGString() - pobiera wpisywany przez użytkownika tekst.

2.2 Game.java

Jest to najważniejsza klasa w aplikacji. Odpowiada ona za przeprowadzenie całej rozgrywki, czyli zawiera kluczowe części algorytmu. Klasa ta składa się z następujących metod:

void generatePrisoners() - metoda generuje początkową populację więźniów (obiekty klasy **Prisoner**). Liczba więźniów ograniczona jest przez użytkownika. Podczas generowania więźnia przydzielany jest mu unikalny numer oraz losowana jest jego pierwsza decyzja.

void runGame() - metoda wywoływana z klasy **Main**. Jej zadaniem jest przeprowadzenie całego algorytmu. W jej wnętrzu wywoływane są metody odpowiadające za przeprowadzenie pojedynków, krzyżowanie, mutację oraz zliczanie wyników.

void cross() - metoda krzyżująca. Losuje ona dwoje więźniów, którzy stają się rodzicami. Więźniowie podczas losowania mają przypisane wagi, które dają im większe szanse na reprodukcję. Jedna para rodziców generuje dwoje dzieci. Zrodzone dzieci zostają dodane do nowej generacji. Jeśli krzyżowanie nie dojdzie do skutku to wylosowani rodzice zostają dodani do nowej generacji.

void mutate() - metoda odpowiadająca za wywołanie mutacji.

void calculateGeneration() - metoda odpowiadająca za rozegranie wszystkich gier w danej generacji.

void duel(Prisoner firstPrisoner, Prisoner secondPrisoner) – metoda odpowiadająca za jeden pojedynek. Jako parametry przyjmowani są dwaj więźniowie pomiędzy którymi rozgrywany jest pojedynek. Wyniki pojedynku są zależne od poprzedniej odpowiedzi danego więźnia.

void printScroes(boolean sort) – metoda wypisująca na ekranie wyniki. Parametr **sort** odpowiada za sortowanie wyników od największego do najmniejszego.

void printProgress() - metoda wypisująca postęp pracy algorytmu.

int getNumberOfRounds() - metoda zwracająca liczbę rozgrywanych rund.

2.3 Decison.java

Klasa reprezentująca decyzję podejmowaną przez więźnia. Klasa składa się z następujących metod oraz pól:

enum Type – typ wyliczeniowy upraszczający zarządzanie decyzjami. Zawiera w sobie elementy *Cooperate* oraz *Defect*.

static Type generateRandomDecision() - statyczna metoda generująca w pełni losową decyzję.

static Type generateDecision(int cooperationPropability) – statyczna metoda generująca decyzję na podstawie podanego prawdopodobieństwa.

2.4 Prisoner.java

Klasa reprezentująca jednego więźnia. Klasa składa się z następujących metod:

static Prisoner getRandomPrisoner(Collection<Prisoner> prisonersToChoose, double maxScore) – metoda zwracająca losowego więźnia uwzględniając liczbę punktów jako wagę.

static LinkedHashMap<String, Prisoner> getCrossedPrisoner(Prisoner father, Prisoner mother, int currentPopulation) – metoda odpowiadająca za krzyżowanie dwóch więźniów. Najpierw losowana jest liczba typu double, która określa jaki procent cechy rodzica A oraz B będzie miało dziecko (przy wylosowaniu 0,75 pierwsze dziecko będzie miało 75% cechy rodzica A oraz 25% cechy rodzica B. Drugie dziecko będzie miało 75% cechy rodzica B i 25% rodzica A). Po wylosowaniu osobnych wartości dla każdej cechy (większa losowość) wygenerowane dzieci są zwracane jako mapa.

void mutate() - metoda mutująca więźnia na rzecz którego została wywołana. Mutacja polega na pomnożeniu obecnej wartości cechy przez losową liczbę typu double oraz dodanie losowej liczny całkowitej z przedziału od 0 do 30.

void generateProbabilities() - metoda generująca prawdopodobieństwa odpowiedzi dla więźnia na rzecz którego została wywołana.

Pozostałe metody klasy są mało znaczące z punktu widzenia algorytmu (są to zwykłe gettery oraz settery).

2.5 Probabilities.java

Klasa przetrzymująca wartości prawdopodobieństwa dalszej współpracy po konkretnej nagrodzie. Klasa składa się z następujących metod:

static boolean getRandom(int propability) – statyczna metoda losująca prawdopodobieństwo z podanego przedziału. Zwraca **true** jeśli wylosowane prawdopodobieństwo jest mniejsze niż podany argument. W przeciwnym wypadku zwraca **false**.

Pozostałe metody klasy są mało znaczące z punktu widzenia algorytmu (są to zwykłe gettery oraz settery).

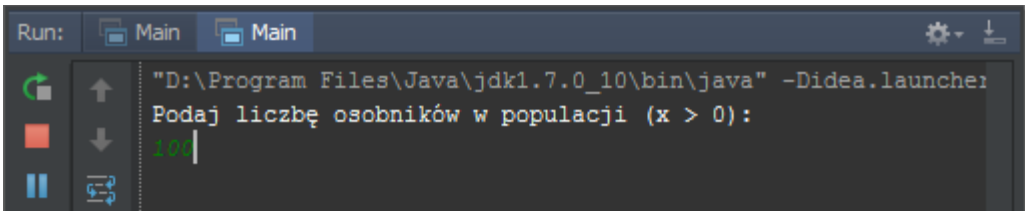
2.6 Rewards.java

Jest to typ wyliczeniowy przetrzymujący odpowiednie nagrody (Temptation, Reward, SuckersPayoff, Punishment) oraz odpowiadającą im liczbę punktów (odpowiednio: 5, 3, 0, 1).

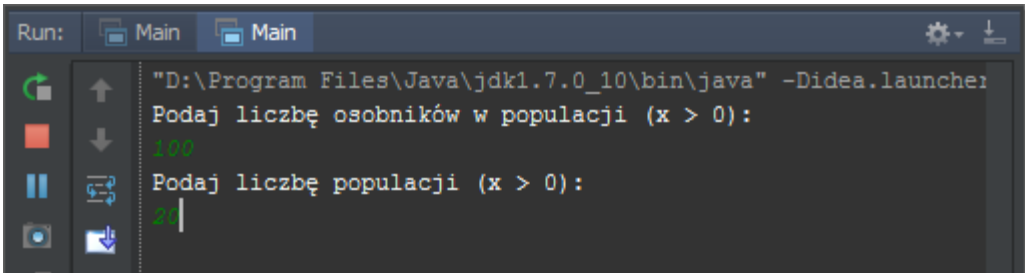
3. Opis algorytmu.

1. Podanie liczebności populacji (rys. 4.1).
2. Podanie liczby generacji (rys. 4.2).
3. Podanie liczby rund (rys. 4.3).
4. Podanie prawdopodobieństwa mutacji (rys. 4.4).
5. Podanie prawdopodobieństwa krzyżowania (rys. 4.5).
6. Wybór czy wyniki mają być pokazywane co generację (rys. 4.6).
7. Wybór czy wyniki mają być zerowane co generację (rys. 4.6).
8. Rozpoczęcie gry.
9. Generacja początkowej populacji.
10. Przeprowadzanie pojedynku dwóch wylosowanych graczy.
11. Krzyżowanie.
12. Mutowanie.
13. Powrót do kroku aż algorytm nie wykona pojedynków wszystkich graczy (każdy z każdym).
14. Wydruk wyników (rys. 4.8).

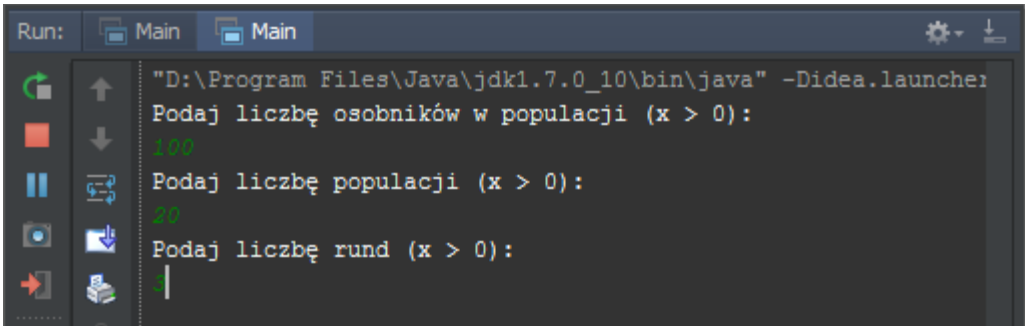
4. Zrzuty ekranu.



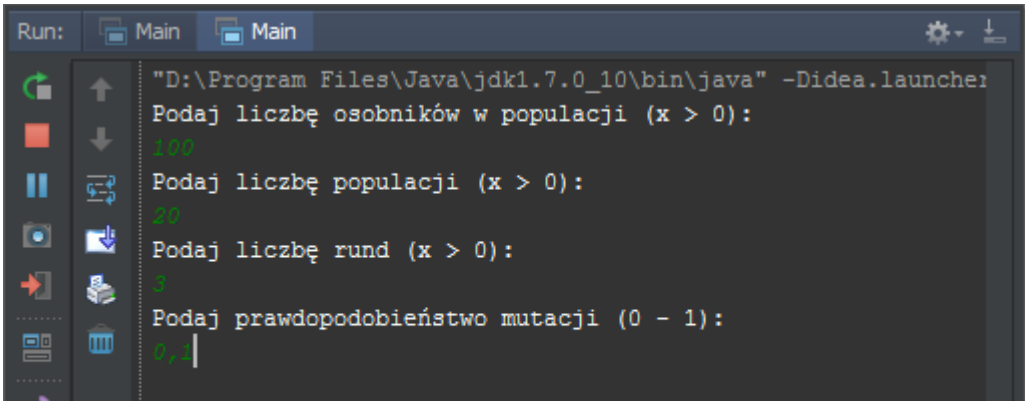
Rys. 4.1. Określenie wielkości populacji.



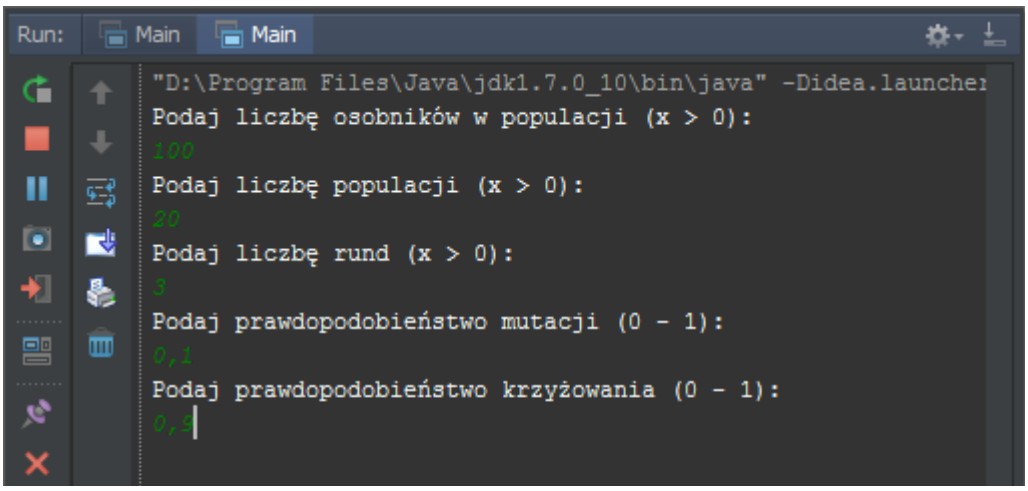
Rys. 4.2. Określenie liczby populacji (generacji)



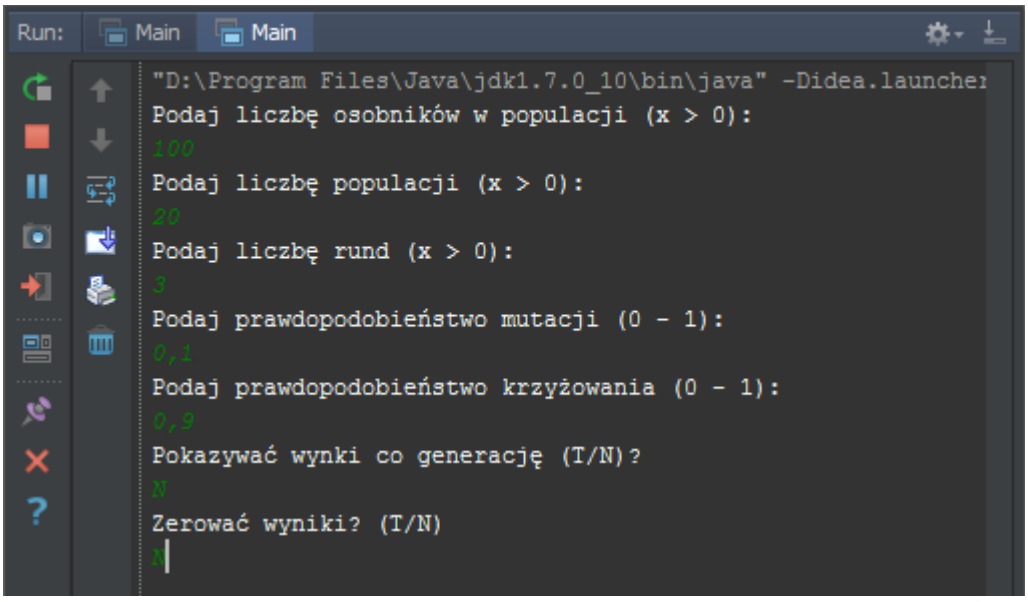
Rys. 4.3. Określenie liczby rund



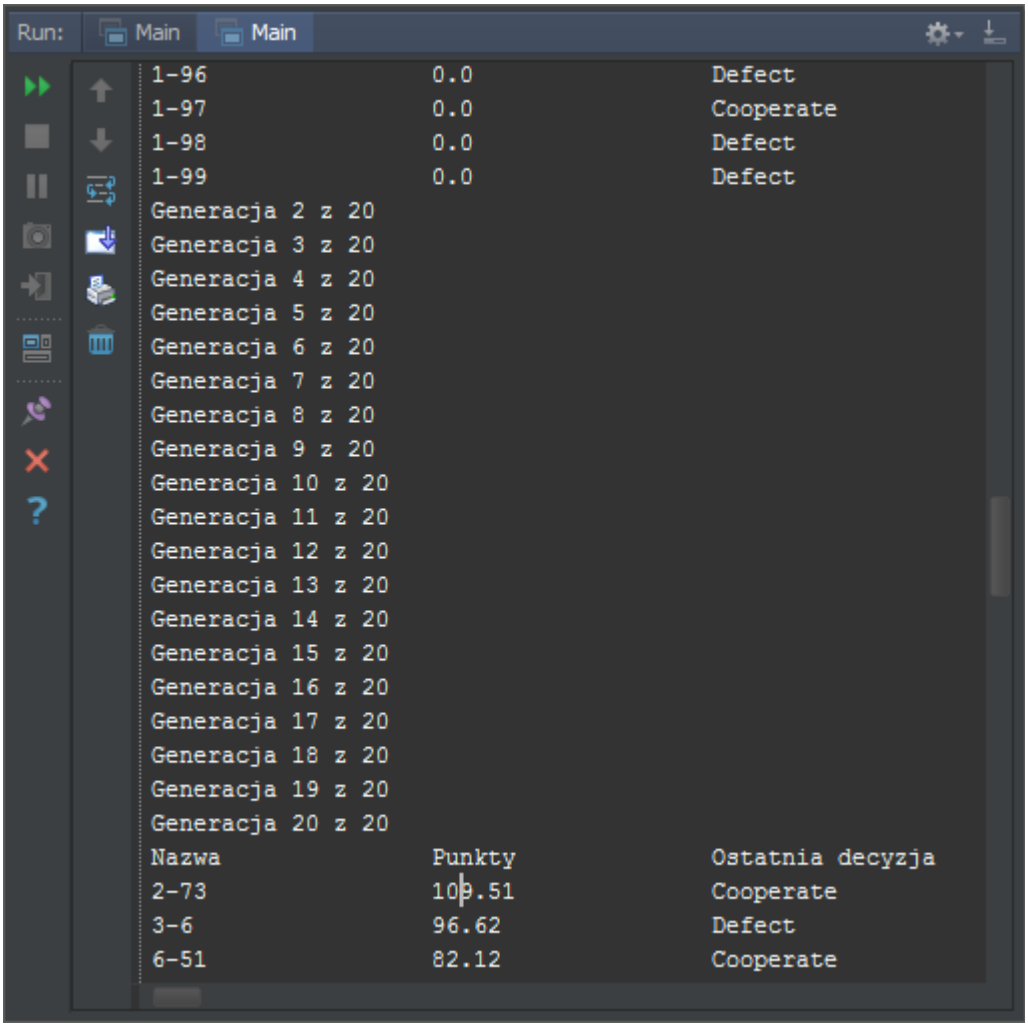
Rys. 4.4. Określenie prawdopodobieństwa mutacji



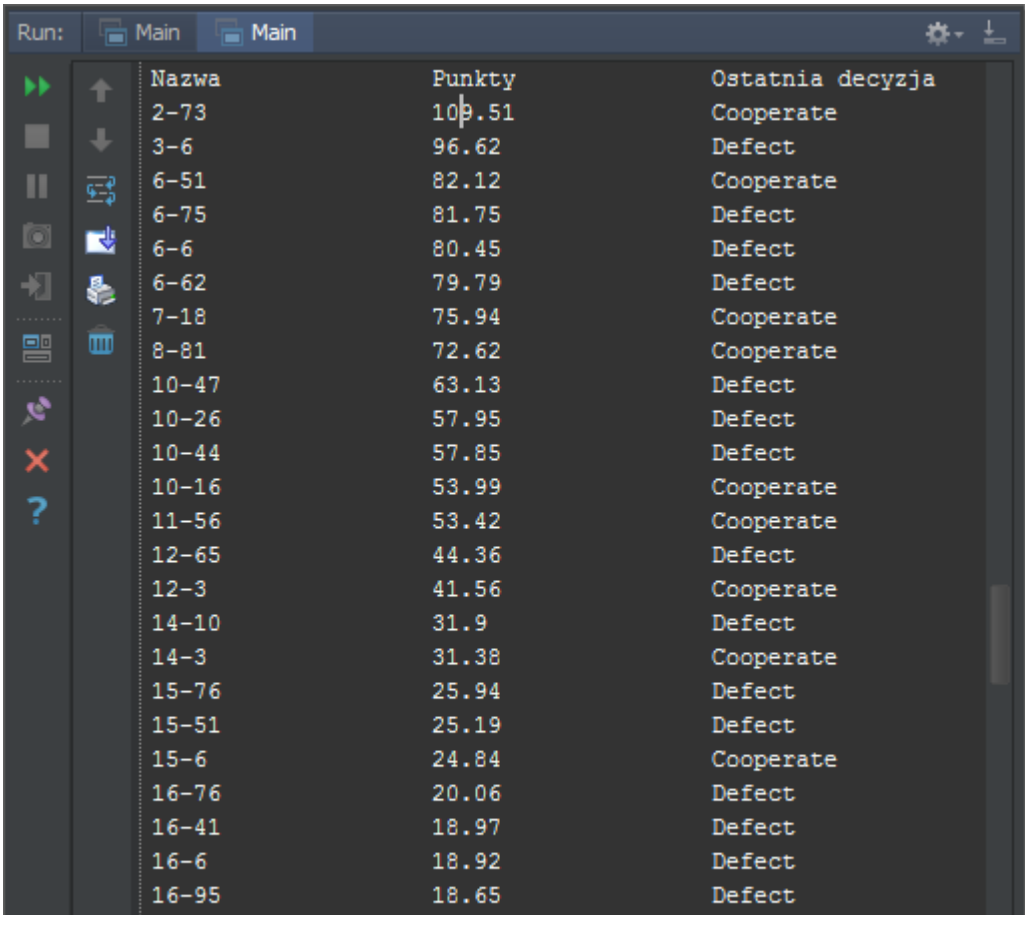
Rys. 4.5. Określenie prawdopodobieństwa krzyżowania



Rys. 4.6. Określenie czy tablica punktacji jest wyświetlana co generację oraz czy zerować punkty osobników po każdej generacji



Rys. 4.7. Tworzenie kolejnych generacji



Rys. 4.8. Wyświetlenie ostatecznych wyników

5. Wnioski:

Na podstawie testów i obserwacji można zauważyć, że wraz ze wzrostem populacji wyniki uśredniają się. Oznacza to, że słabsze osobniki są eliminowane z populacji, krzyżowane są natomiast osobniki silniejsze, co powoduje powstanie mocniejszego pokolenia.

Podczas przeprowadzania testów można było zauważyć jak duży wpływ na wyniki i znalezienie najlepszego gracza ma krzyżowanie i mutacja. Procesy te zapewniają większe urozmaicenie osobników dzięki czemu tworzą większe szanse na powstanie najlepiej dostosowanego gatunku, który może być wstanie zdominować całą populację.

Przedstawiona implementacja tego algorytmu jest uproszczeniem procesów zachodzących w naturze. Wierne odwzorowanie tych procesów wydaje się niemożliwe z powodu ograniczeń w zakresie losowości w maszynach cyfrowych.