Techniki Komponentowe

Projekt 7: Cellular evolutionary algorithm

Autorzy: Wojciech Buś, Wojciech Klemens, Paweł Mendroch

1. Wstęp

Celem naszego systemu jest utworzenie modularnego oprogramowania implementującego algorytm komórkowej ewolucji pozwalającego na szukanie optymalnych rozwiązań zadanych przez użytkownika problemów.

Technologia używana w projekcie to Kotlin Coroutines, pozwalająca na tworzenie dużego systemu komunikacji aktorów porozumiewających się ze sobą.

2. Parametry wejściowe programu

Liczba aktorów, liczba sentineli (aktorów raportujących do loggera o najlepszym napotkanym genotypie) oraz liczba iteracji.

3. Genotypy

W celu zdefiniowania konkretnego problemu optymalizacyjnego dla naszego systemu użytkownik musi zaimplementować klasę genotyp. Obiekty tej klasy odpowiadają potencjalnym argumentom optymalizowanej funkcji, zaś proces szukania jej ekstremum odbywać się będzie poprzez reprodukcję aktorów z różnymi genotypami w celu maksymalizacji ich jakości (fitness, wartość optymalizowanej funkcji).

Przykładowe implementacje problemów optymalizacyjnych utworzone przez nas to:

- maksymalizacja pola prostokąta o określonym obwodzie - genotyp to pojedyncza liczba rzeczywista.
- funkcja Beale'a - genotyp to krotka dwóch liczb rzeczywistych.
- funkcja Ackleya - genotyp to krotka dwóch liczb rzeczywistych.

W każdym z powyższych przypadków algorytm reprodukcji dwóch genotypów zwracał ich średnią arytmetyczną zmodyfikowaną o pewną losową wartość (mutację).

Ponadto implementujemy wrapper bestGenotype, który służy do przechowywania informacji o najlepszym dotychczasowo wyliczonym genotypie wraz z timestampem jego wyznaczenia.

4. Aktorzy

Główną jednostką w programie są aktorzy (odpowiedniki komórek/osobników), odpowiedzialni za przechowywanie, modyfikowanie i rozprzestrzenianie potencjalnych rozwiązań danego problemu (genotypów). Każdy aktor jest w pełni autonomiczny i może działać równolegle do pozostałych. Na klasę aktora składa się:

- jego sąsiedztwo, czyli kolekcja innych aktorów, z którymi może się on komunikować,
- jego genotyp (reprezentacja przykładowego rozwiązania danego problemu).

Ponadto, aktor (jako element siatki aktorów) pełni funkcję przechowywania i rozprzestrzeniania informacji na temat najlepszych napotkanych przez sieć rozwiązań. Jeżeli należy do grupy sentineli (reprezentantów), to odpowiadając na zapytania loggera zwraca tę informację w wiadomości zwrotnej.

- **tworzenie aktorów**

Tworzenie aktorów odbywa się w głównym wątku programu. Aktorzy zostają tymczasowo umieszczeni na siatce (tablicy) MxN w celu ułatwienia zdefiniowania sąsiedztwa między nimi. Po wstępnej konfiguracji aktorzy działają samodzielnie aż do momentu zakończenia obliczeń, po przejściu odpowiedniej liczby iteracji. Ponadto, spośród utworzonych aktorów zostaje wybranych x (argument uruchomieniowy) sentineli, którzy zostają wyznaczeni do komunikacji z aktorem logującym najlepszy fitness.

- **działanie aktora**

Aktor ma dwa główne zadania:

1. wysyłanie wiadomości do sąsiadów oraz loggera
2. odbieranie wiadomości od sąsiadów i loggera oraz odpowiednia ich obsługa

W związku z tym aktor został wewnętrznie podzielony na dwie coroutines, jako że wysyłanie wiadomości i odbieranie jest operacją blokującą, a chcemy by operacje te działy się równolegle.

Każdy aktor ma swój własny kanał, na który przesyłane są wiadomości do niego skierowane. Jedynym globalnym kanałem jest kanał aktora logującego, na który posyłane są odpowiedzi na zapytania o najlepszy napotkany genotyp.

Coroutine odpowiedzialna za odbieranie i obsługę wiadomości nie posyła odpowiedzi na odebrane pingi, gdyż blokowałoby to komunikację. Zamiast tego umieszcza odpowiedź w odpowiedniej kolejce, którą następnie obsłuży coroutine odpowiedzialna za wysyłanie wiadomości.

Coroutine odpowiedzialna za wysyłanie wiadomości priorytetyzuje odpowiedzi na pingi, reprodukcję oraz zapytania loggera. W sytuacjach, w których nie ma wiadomości do obsłużenia, pinguje wszystkich swoich sąsiadów inicjując komunikację i swój własny proces reprodukcji.

- **reprodukcja**

Proces rozmnażania aktorów przebiega w sposób następujący: Aktor wysyła serię pingów do swoich sąsiadów, oni odsyłają ponga ze swoimi genotypami. Następnie aktor wybiera z listy otrzymanych genotypów genotyp, z którym chce się rozmnażać i z nim kopuluje (wyznacza genotyp dziecka w oparciu o genotypy swój i partnera). Następnie aktor wybiera

genotyp, który chce zastąpić - może być to jeden z sąsiadów lub on sam. Jeżeli wybrał siebie to podmienia swój genotyp na nowy (jeżeli jego obecny genotyp jest gorszy), a jeśli kogoś innego, to wysyła do niego prośbę o zastąpienie genotypu. Aktor, który otrzymał podane żądanie sprawdza, czy otrzymany genotyp jest lepszy od aktualnie posiadanego - jeśli tak, to dokonuje podmiany.

5. **Wiadomości**

Komunikacja między poszczególnymi aktorami oraz między loggerem i sentinelami wymaga 6 typów wiadomości:

1. Ping - jest to zapytanie aktora o aktualny genotyp jego sąsiada. Wysyłane zazwyczaj seriami do wszystkich sąsiadów. Zawiera kanał, na który sąsiad ma wysłać odpowiedź. Ponadto rozprzestrzenia w sieci informacje o najlepszym napotkanym genotypie.
2. Pong - odpowiedź na ping, zawiera własny genotyp oraz kanał na odpowiedź w przypadku nadchodzącej wiadomości Replace. Również przesyła informację o najlepszym poznanym genotypie.
3. Replace - wiadomość wysyłana do danego aktora, jeżeli dziecko któregoś z sąsiadów ma zająć jego miejsce. Zawiera nowy genotyp do podmiany w miejsce aktualnego.
4. Logger ping - zapytanie o najlepszy napotkany genotyp.
5. Logger pong - odpowiedź na ping loggera z najlepszym znanym genotypem.
6. Finish - aktor po przejściu danej liczby iteracji (argument uruchomieniowy) posyła do loggera wiadomość o zakończeniu pracy - Logger jest aktorem na który główny wątek czeka, więc po otrzymaniu M * N wiadomości Finish sam zakończy pracę

6. **Modularyzacja**

W celu zwiększenia elastyczności i reużywalności kodu staraliśmy się zmaksymalizować jego modularyzację, korzystając tam gdzie to możliwe z interfejsów - dzięki temu użytkownik naszego systemu jest w stanie wykorzystać go do szukania rozwiązań potencjalnie dowolnych problemów na różne sposoby, o ile zaimplementowane przez niego komponenty będą spełniać wymagania narzucone przez nasze interfejsy.

- **Interfejs IGenotype**

```
interface IGenotype{
    fun fitness() : Double
    fun reproduce(other : IGenotype) : IGenotype
}
```

Opisuje metody, które muszą implementować konkretne genotypy. Stanowi więc niejako jedyne ograniczenie na klasę problemów, które jest w stanie rozwiązać nasz system.

Są to:

- double fitness() - zwraca jakość danego rozwiązania (genotypu). System będzie dążył do jego maksymalizacji.
- IGenotype reproduce(IGenotype other) - definiuje proces reprodukcji (wraz z ewentualnymi mutacjami) dwóch genotypów i zwraca ich potomka. Argument *other* tej metody oraz obiekt na którym została wywołana powinny być obiektami tej samej klasy.

Zaimplementowany genotyp natomiast wystarczy podmienić w kodzie klasy Actor:

```
class Actor(val id: Int, private val logChannel: Channel<IMessage>, private val nIter: Int) {
    val actorChannel = Channel<IMessage>()
    private var neighbours: MutableList<Channel<IMessage>> = ArrayList()
    private var reproduceChooser: GenotypeChooser = GenotypeBestChooser()
    private var removeChooser: GenotypeChooser = GenotypeWorstChooser()
    private var genotype: IGenotype = GenotypeExample2()
```

Nasze przykładowe implementacje:

- Maksymalizacja pola prostokąta o obwodzie równym 2 i jednym boku długości x:

```
class GenotypeExample1(xInitial: Double = Math.random()) : IGenotype {
    private val x = xInitial

    override fun fitness() = x * (1 - x)
    override fun reproduce(other: IGenotype): IGenotype {
        other as GenotypeExample1

        val xNew = (x + other.x) / 2
        val diff = abs( x: x - other.x)
        val randomOffset = (Math.random() - 0.5) * diff

        return GenotypeExample1( xInitial: xNew + randomOffset)
    }

    override fun toString(): String {
        return x.toString()
    }
}
```

- funkcja Beale’a:

```
class GenotypeExample2(xInitial: Double = randrange( a: -4.5,  b: 4.5), yInitial: Double = randrange( a: -4.5,  b: 4.5)) : IGenotype {
    private val x = xInitial
    private val y = yInitial

    override fun fitness(): Double {
        return -(1.5 - x + x * y) * (1.5 - x + x * y) - (2.25 - x + x * y * y) *
                (2.25 - x + x * y * y) - (2.625 - x + x * y * y * y) * (2.625 - x + x * y * y * y)
    }

    override fun reproduce(other: IGenotype): IGenotype {
        other as GenotypeExample2

        val xNew = (x + other.x) / 2
        val xRandomOffset = (Math.random() - 0.5) * abs( x: x - other.x) * 10
        val yNew = (y + other.y) / 2
        val yRandomOffset = (Math.random() - 0.5) * abs( x: x - other.x) * 10

        return GenotypeExample2( xInitial: xNew + xRandomOffset,  yInitial: yNew + yRandomOffset)
    }

    override fun toString(): String {
        return "($x, $y)"
    }
}
```

- funkcja Ackleya

```
class GenotypeExample3(xInitial: Double = randrange( a: -5.0,  b: 5.0), yInitial: Double = randrange( a: -5.0,  b: 5.0)) : IGenotype {
    private val x = xInitial
    private val y = yInitial

    override fun fitness(): Double {
        return 20 * exp( x: -0.2 * sqrt( x: 0.5 * (x * x + y * y))) +
                exp( x: 0.5*(cos( x: 2*PI*x) + cos( x: 2*PI*y))) - exp( x: 1.0) - 20
    }

    override fun reproduce(other: IGenotype): IGenotype {
        other as GenotypeExample3

        val xNew = (x + other.x) / 2
        val xRandomOffset = (Math.random() - 0.5) * abs( x: x - other.x) * 10
        val yNew = (y + other.y) / 2
        val yRandomOffset = (Math.random() - 0.5) * abs( x: x - other.x) * 10

        return GenotypeExample3( xInitial: xNew + xRandomOffset,  yInitial: yNew + yRandomOffset)
    }

    override fun toString(): String {
        return "($x, $y)"
    }
}
```

- **Klasa abstrakcyjna ActorSpawner**

```
abstract class ActorSpawner(
    protected val m: Int,
    protected val n: Int,
    private val nIter: Int
) {
```

Opisuje schemat działania komponentu odpowiedzialnego za utworzenie odpowiedniej liczby (MxN) aktorów i zdefiniowania sąsiedztw między nimi na początku uruchomienia całego systemu. Metody odpowiedzialne za inicjalizację obiektów aktorów są tu już

zaimplementowane, jedyną metodą abstrakcyjną tej klasy jest metoda attachNeighbours(), której konkretne implementacje powinny definiować sposób łączenia aktorów w sąsiedztwa.

Zaimplementowaną klasę tworzącą sąsiedztwo podmieniamy w głównej klasie Main, w funkcji uruchamiającej aktorów.

```
suspend fun launchActors(m: Int, n: Int, x: Int, nIter: Int) {
    val logChannel = Channel<IMessage>()
    val actorSpawner = CrossActorSpawner(m, n, nIter)
```

3 przykłady utworzone przez nas:

- CrossActorSpawner

Sąsiedzi tworzeni w formie krzyża:

```
class CrossActorSpawner(m: Int, n: Int, nIter: Int) : ActorSpawner(m, n, nIter) {
    override fun attachNeighbours() {
        for (i in 0 until m) {
            for (j in 0 until n) {
                val neighbours: MutableList<Channel<IMessage>> = ArrayList()

                if (i - 1 >= 0)
                    neighbours.add(actorGrid[i - 1][j].actorChannel)
                if (i + 1 <= m - 1)
                    neighbours.add(actorGrid[i + 1][j].actorChannel)
                if (j - 1 >= 0)
                    neighbours.add(actorGrid[i][j - 1].actorChannel)
                if (j + 1 <= n - 1)
                    neighbours.add(actorGrid[i][j + 1].actorChannel)

                actorGrid[i][j].setNeighbours(neighbours)
            }
        }
    }
}
```

- LittleSquareActorSpawner

Sąsiedzi tworzeni w formie kwadratów wokół danego aktora:

```
class LittleSquareActorSpawner(m: Int, n: Int, nIter: Int) : ActorSpawner(m, n, nIter) {
    override fun attachNeighbours() {
        for (x in 0 until m) {
            for (y in 0 until n) {
                val neighbours: MutableList<Channel<IMessage>> = ArrayList()

                for (i in -1..1) {
                    for (j in -1..1) {
                        if (i == 0 && j == 0) {
                            continue
                        }
                        if (isValidCoordinates( x: x + i,  y: y + j)) {
                            neighbours.add(actorGrid[x + i][y + j].actorChannel)
                        }
                    }
                }

                actorGrid[x][y].setNeighbours(neighbours)
            }
        }
    }

    private fun isValidCoordinates(x: Int, y: Int): Boolean {
        return (x in 0 until m) && (y in 0 until n)
    }
}
```

- BigSquareActorSpawner

Struktura identyczna jak w LittleSquare, jednak zamiast kwadratu 3x3 bierzemy pod uwagę sąsiedztwo w kwadracie 5x5:

```
for (i in -2..2) {
    for (j in -2..2) {
        if (i == 0 && j == 0) {
            continue
        }
        if (isValidCoordinates( x: x + i,  y: y + j)) {
            neighbours.add(actorGrid[x + i][y + j].actorChannel)
        }
    }
}
```

- **Interfejs IGenotypeChooser**

```
interface IGenotypeChooser {
    fun choose(genotypes: List<IGenotype>) : Int
}
```

Opisuje metodę, którą muszą implementować komponenty odpowiedzialne za wybór partnera do reprodukcji / wybór aktora do wymiany.

- int choose(List<IGenotype> genotypes) - metoda przyjmuje jako argument pewną pulę genotypów i definiuje mechanizm wyboru jednego z nich. Zwraca liczbę całkowitą będącą indeksem wybranego elementu.

Zaimplementowaną klasę następnie należy podmienić w kodzie klasy Actor zarówno dla tworzenia nowego potomka, jak i wyboru genotypu do podmiany:

```
class Actor(val id: Int, private val logChannel: Channel<IMessage>, private val nIter: Int) {
    val actorChannel = Channel<IMessage>()
    private var neighbours: MutableList<Channel<IMessage>> = ArrayList()
    private var reproduceChooser: GenotypeChooser = GenotypeBestChooser()
    private var removeChooser: GenotypeChooser = GenotypeWorstChooser()
```

Zaimplementowane przez nas przykładowe klasy:

- GenotypeBestChooser - wybór najlepszego genotypu z danej listy

```
class GenotypeBestChooser : IGenotypeChooser {
    override fun choose(genotypes: List<IGenotype>) : Int {
        var bestIndex = -1
        var bestFitness = -Double.MAX_VALUE

        for (i in genotypes.indices) {
            if (genotypes[i].fitness() > bestFitness) {
                bestFitness = genotypes[i].fitness()
                bestIndex = i
            }
        }

        return bestIndex
    }
}
```

- GenotypeRandomChooser - wybór losowego genotypu

```
class GenotypeRandomChooser : IGenotypeChooser {
    override fun choose(genotypes: List<IGenotype>) : Int {
        return Random.nextInt( from: 0, genotypes.size)
    }
}
```

- GenotypeWorstChooser - wybór najgorszego genotypu z listy

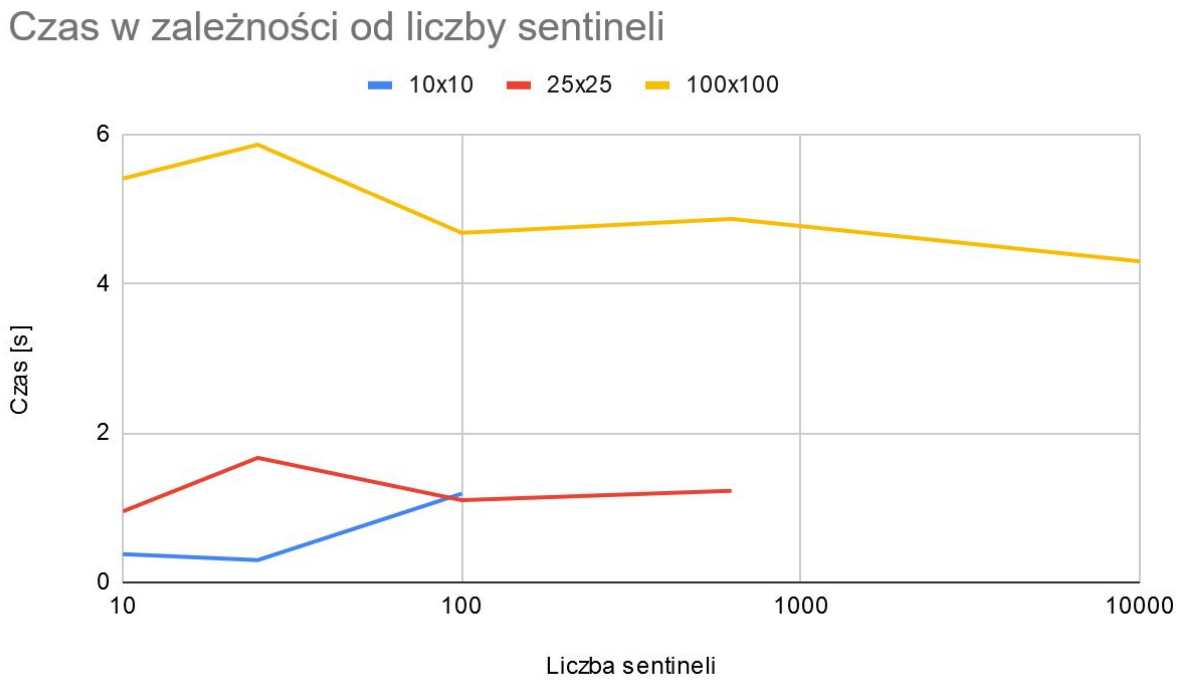
```
class GenotypeWorstChooser : IGenotypeChooser {
    override fun choose(genotypes: List<IGenotype>) : Int {
        var worstIndex = -1
        var worstFitness = Double.MAX_VALUE

        for (i in genotypes.indices) {
            if (genotypes[i].fitness() < worstFitness) {
                worstFitness = genotypes[i].fitness()
                worstIndex = i
            }
        }

        return worstIndex
    }
}
```

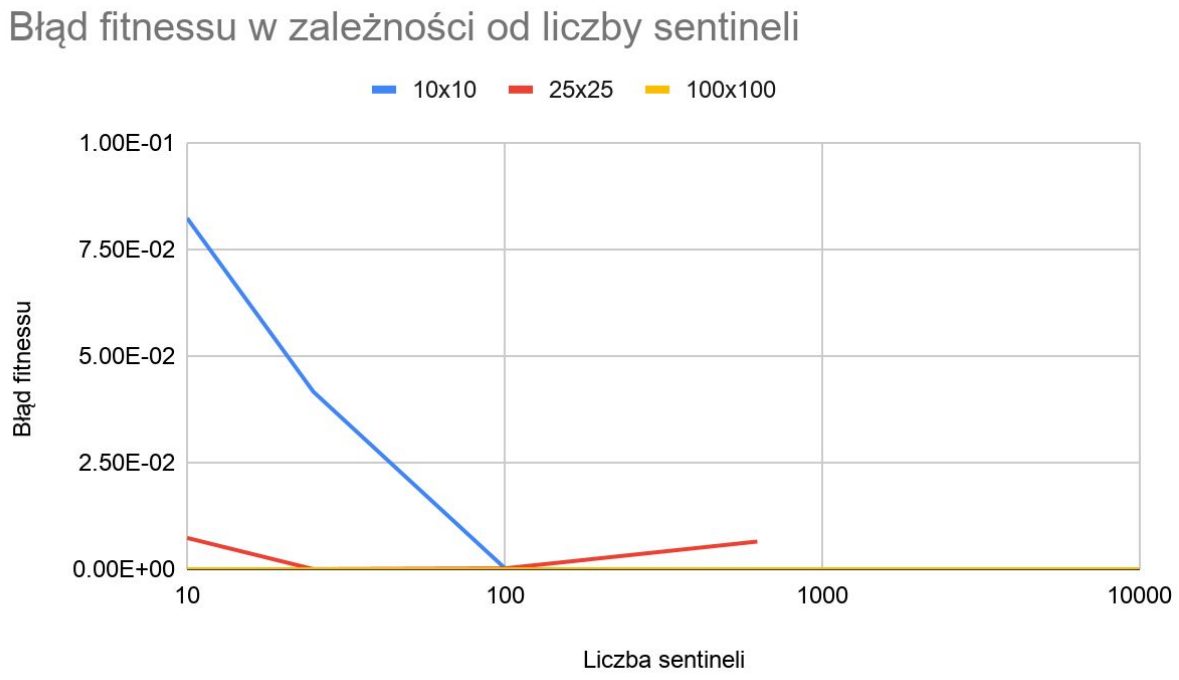
7. Testy

W poniższym punkcie pokażemy wyniki testów, jakie możemy uzyskać poprzez zamianę różnych modułów w naszym programie. Będzie to między innymi liczba sentineli, rozmiar siatki, na którym znajdują się nasi aktorzy, rodzaj genotypu, rodzaj sąsiedztwa, oraz rodzaj wyboru aktora do reprodukcji czy też wymiany.

Poniższe testy wykonywane dla GenotypeExample2 - funkcji Beale’a.

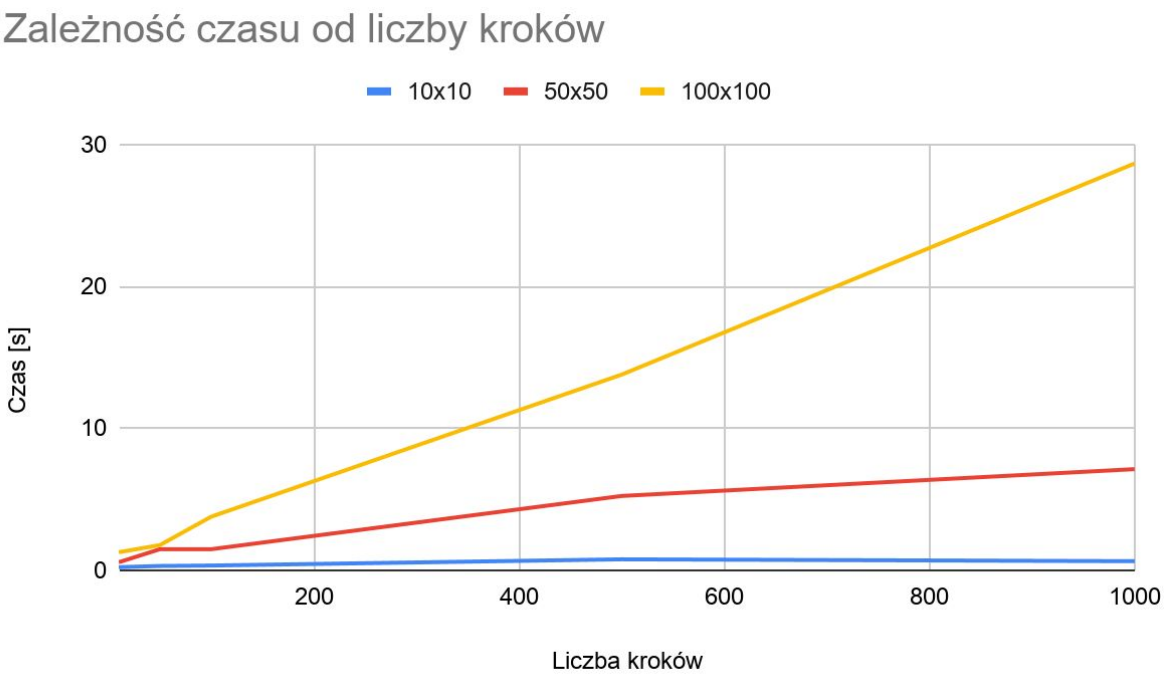
- **Zależność czasu oraz fitnessu od liczby sentineli**

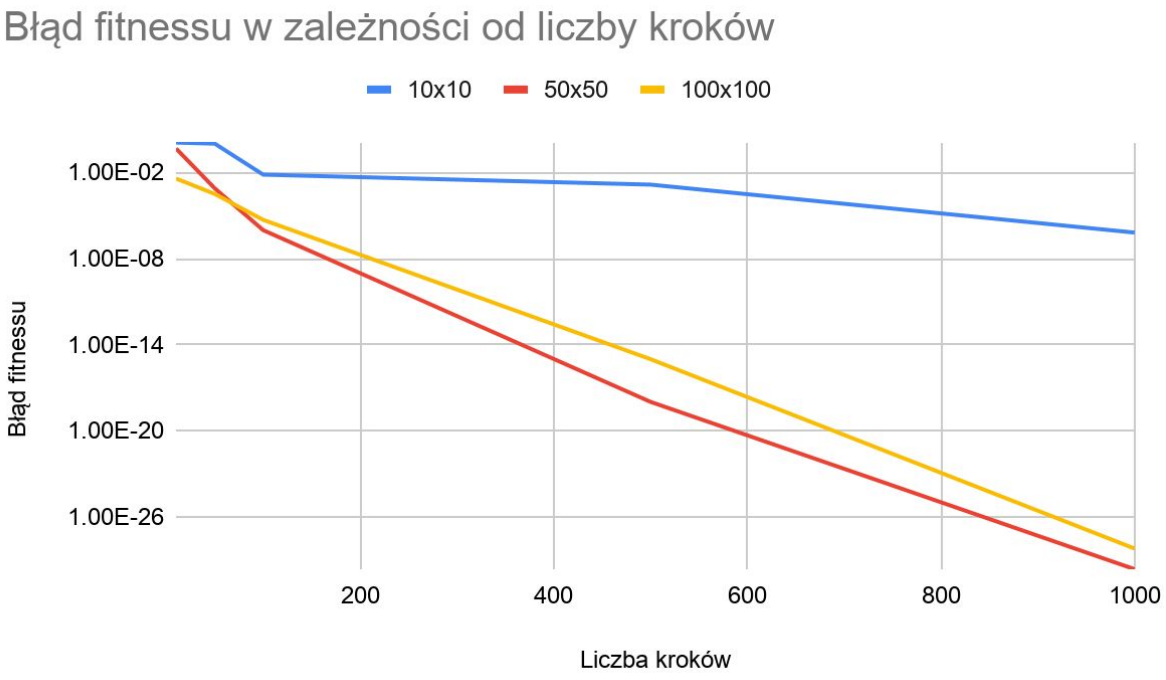




Jak widać na powyższych wykresach, czas jest w niewielkim stopniu zależny od liczby sentineli, mimo intuicji mówiącej, że zwiększenie tej liczby powinno spowolnić program. Fitness natomiast jest widocznie lepszy przy większej ich liczbie, gdyż szybciej otrzymujemy informację w loggerze. Ma to jednak znikome znaczenie przy większej liczbie iteracji czy aktorów,, gdzie dokładność fitnessu tak czy siak zbliża się do dokładności liczb zmiennoprzecinkowych w Kotlinie.

- **Zależność fitness od liczby iteracji**



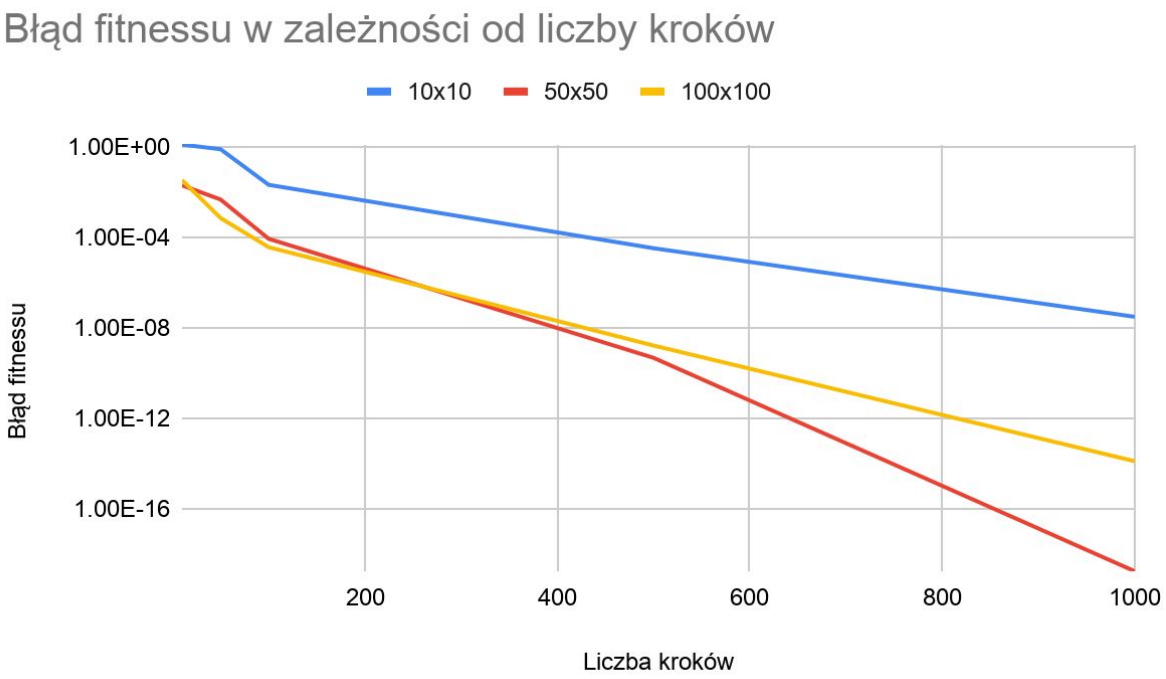


Jak widać, liczba kroków wpływa mniej więcej liniowo na czas działania programu, co zgadza się z intuicją. Wpływa również znacząco na poprawienie jakości wyników, jednakże podobnie jak wcześniej względnie szybko jesteśmy w stanie osiągnąć dokładność porównywalną z dokładnością użytych typów liczbowych.

- **Fitness w zależności od wyboru i podmiany genotypów**

Uprzednio badaliśmy wyniki dla ustawień reprodukcji zakładających wybór najlepszego partnera i podmianę najgorszego. Następnie chcemy sprawdzić jak zachowają się wyniki, gdy zarówno reprodukcja jak i zastąpienie dotyczyć będzie losowego aktora.

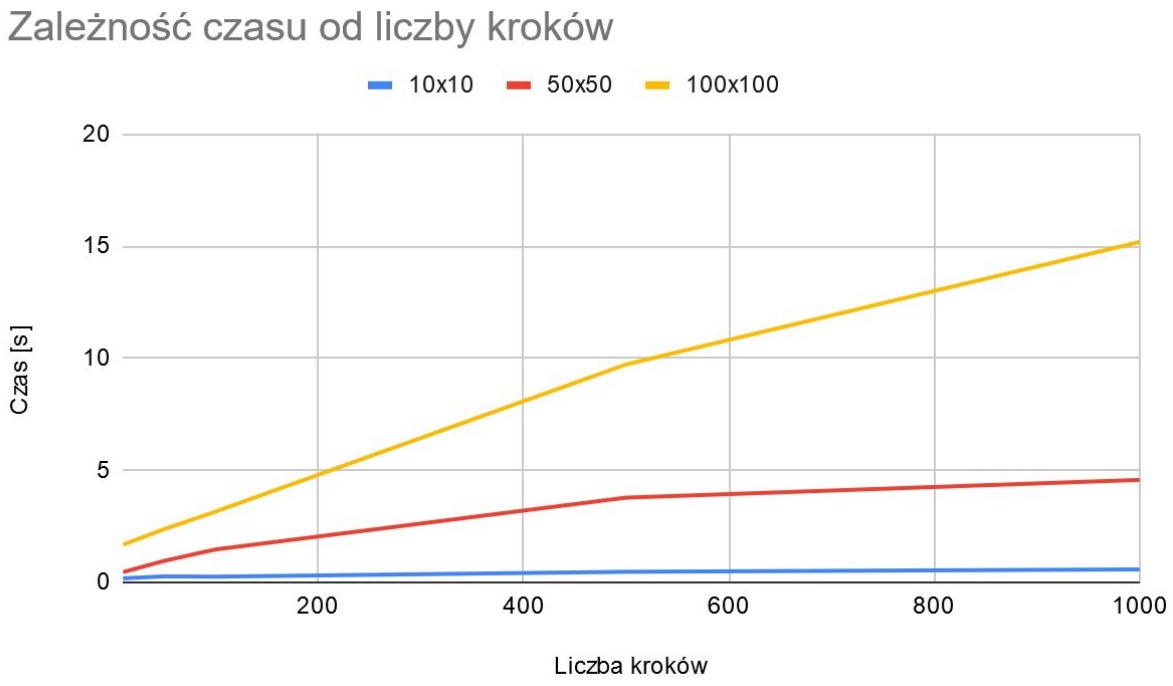
```
private var reproduceChooser: IGenotypeChooser = GenotypeRandomChooser()
private var removeChooser: IGenotypeChooser = GenotypeRandomChooser()
```

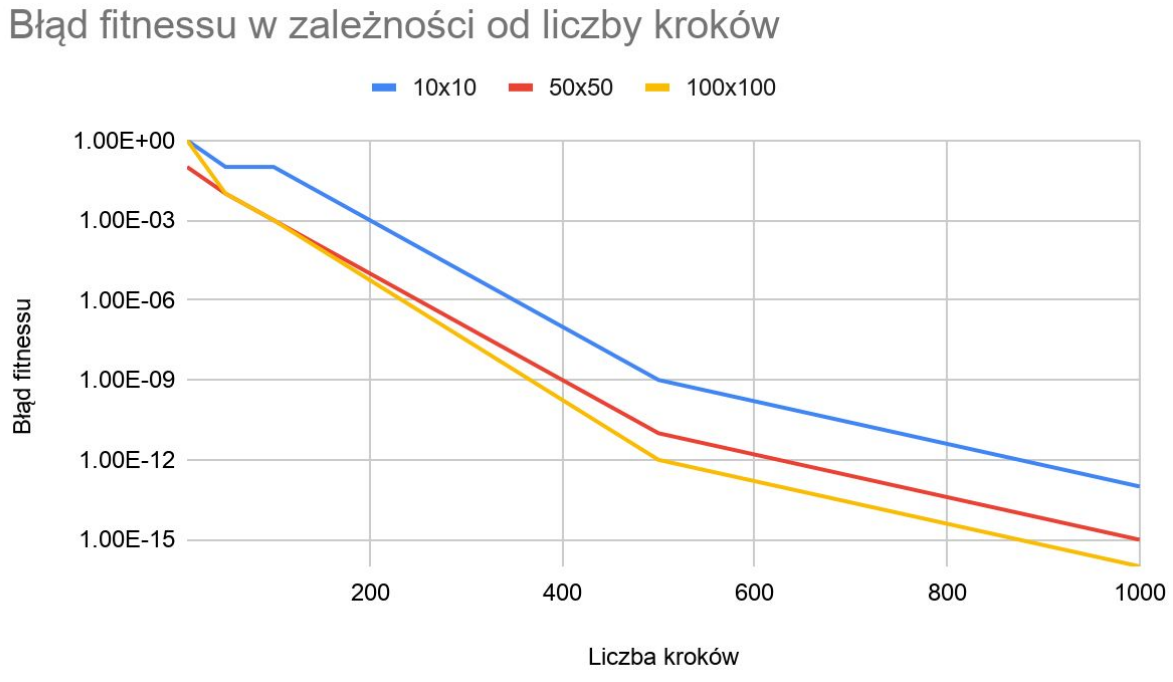


Mimo że początkowo różnice między wyborami są niewielkie, to dla większej liczby kroków widzimy, że fitness dochodzi do błędu rzędu góra 1e-19, gdzie wybierając najlepsze przypadki udało się dojść do 1e-30.

- **Fitness w zależności od wyboru i podmiany genotypów**

Podobną analizę możemy przeprowadzić dla genotypu, który jest funkcją Ackleya (dla wyboru losowego genotypu z sąsiedztwa, oraz zastępowanie losowego genotypu z sąsiedztwa):

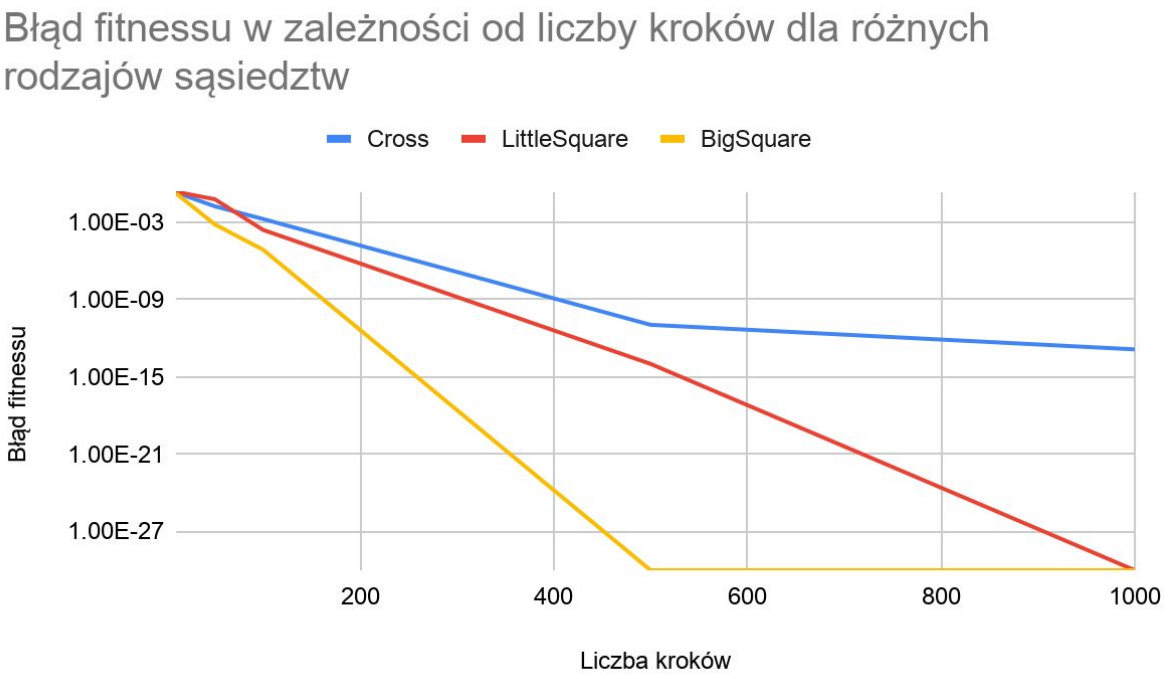




Jak widzimy, otrzymaliśmy wyniki podobne do poprzednich. Jedyne co musieliśmy zrobić, aby je otrzymać to podmienić genotyp w klasie Aktor.

- **Fitness w zależności od rodzaju sąsiedztwa**

Za genotyp weźmy ponownie funkcję Ackleya. Przeprowadźmy test błędu dla siatki 100x100.



Wartość 1e-30 oznacza tutaj, że znaleziono dokładną szukaną wartość (na tyle, na ile pozwala typ Double w Kotlinie). Jak widzimy, im większe sąsiedztwo, tym szybciej otrzymujemy lepszą dokładność.

8. Kod programu

Repozytorium projektu: [Github]