Algorytmy optymalizacyjne w usłudze sieciowej TCP:

-algorytm genetyczny,

-algorytm roju cząstek.

Daniel Woźniczak

v. 0.3

Spis treści:

Wstęp, Client, Server, ServerThread

By korzystać z aplikacji należy skompilować i uruchomić **Server.java** poleceniem ‘java Server’. Klasa nie przyjmuje argumentów. Następnie do uruchomionego Servera możemy podłączyć dowolną liczbę użytkówników poleceniem ‘java Client <username>’ gdzie username toString z nazwa użytkownika. Po tym poleceniu uruchamia się nowy wątek, na którym działa użytkownik.

Metody:

Server.java

`public static void sendToAll()` – funkcja pozwala na wysłanie wiadomości do wszystkich podłączonych użytkowników. Lista podłączonych użytkowników jest zapisana w ‘List<ServerThread> clients’ – liście obiektów typu ServerThread.

Client.java

`private static void printMenu()` – wyświetla menu w postaci Stringa. Modyfikowalne.

`public static class Reader extends Thread` – klasa rozszerzająca ‘Thread’ z metodą run(). Uruchamiana jako wątek przy starcie ‘Clienta’. Służy do nasłuchiwania, odbierania i wyświetlania wiadomości od Servera.

Funkcjonalność:

Jak wspomniałem wyżej jest to serwer z możliwością logowania użytkowników. Jako klienci możemy wybrać funkcję, którą chcemy optymalizować i jaki algorytm ma to zrobić. Po stronie serwera zapisywane są wszystkie informacje o tym, którą opcję wybrał dany użytkownik.

```
GA - Genetic Algorythm
Wybierz opje:
1. Funkcja Beale'a GA
2. Funkcja Rosenbrocka GA
3. Funkcja Booth'a GA
4. Funkcja Easom'a GA
5. Wyjscie
```

Rys. 1a. przykładowy output Clienta

```
Server up and ready for connections...
user 'user' is now connected to the server
user 'user2' is now connected to the server
Klient 'user2' wybral: 3. funkcja Booth'a GA
Klient 'user' wybral: 1. funkcja Beale'a GA
```

Rys. 1b. przykładowy output Servera

Algorytm genetyczny: wstęp

Problem definiuje środowisko, w którym istnieje pewna populacja osobników. Każdy z osobników ma przypisany pewien zbiór informacji stanowiących jego genotyp, a będących podstawą do utworzenia fenotypu. Fenotyp to zbiór cech podlegających ocenie funkcji przystosowania modelującej środowisko. Innymi słowy - genotyp opisuje proponowane rozwiązanie problemu, a funkcja przystosowania ocenia, jak dobre jest to rozwiązanie.

Genotyp składa się z chromosomów, gdzie zakodowany jest fenotyp i ewentualnie pewne informacje pomocnicze dla algorytmu genetycznego. Chromosom składa się z genów.

Wspólnymi cechami algorytmów ewolucyjnych, odróżniającymi je od innych, tradycyjnych metod optymalizacji, są:

1. stosowanie operatorów genetycznych, które dostosowane są do postaci rozwiązań,
2. przetwarzanie populacji rozwiązań, prowadzące do równoległego przeszukiwania przestrzeni rozwiązań z różnych punktów,
3. w celu ukierunkowania procesu przeszukiwania wystarczającą informacją jest jakość aktualnych rozwiązań,
4. celowe wprowadzenie elementów losowych.

Działanie algorytmu:

1. jest losowana populacja początkowa.
2. Populacja jest selekcjonowana. Najlepiej przystosowane osobniki populacji będą brały udział w procesie reprodukcji.
3. Genotypy wybranych osobników są poddawane krzyżowaniu (łączeniu 2 osobników) i przeprowadzana jest mutacja (niewielki %).
4. Rodzi się następne pokolenie. Aby utrzymać stałą liczbę osobników najlepsze są zostawiane, a najgorsze usuwane.
5. Po zadanej liczbie powtórzeń i wyszukań zwracany jest najlepszy wynik.

Algorytm genetyczny: klasy, metody

Na algorytm genetyczny składają się:
- **Individual.java –** reprezentacja pojedynczego osobnika,
- **Population.java –** reprezentacja populacji złożonej z osobników,
- **GeneticAlgorithms.java –** kod algorytmu genetycznego,
- **Constants.java** – przechowuje zmienne, zdefiniowane na stałe,
- **LineChartE.java –** klasa obsługująca wykresy,
- **App.java –** inicjuje klasy i rozpoczyna program.

Metody i zmienne:

Individual.java

`private int[] genes;` - tablica ‘genów’ osobnika (podawane jako 0 lub 1).

`private Random randomGenerator;` - deklaracja losowego generatora służącego do inicjalizacji losowymi danymi.

`public Individual()` - konstruktor klasy Individual.

`public void generateIndividual()` - generuje losowy zapis 0 i 1 w osobniku.

`public double f(double x, double y)` - zwraca wynik funkcji, którą optymalizujemy. Argumentami są współrzędne x i y.

`public double getFitness()` - zwraca wyliczoną wartość z funkcji `f(double x, double y).`

`public double getFitnessResult(int x)` - zwraca wartość x albo y, w zależności od podanego argumentu (dla 0 zwraca x, dla 1 zwraca y).

`public double genesToDouble(int x)` - zamienia zapis bitowy 0 i 1 na liczbę double. W zależności od podanego argumentu zwraca x (dla 0) lub y (dla 1). W funkcji możemy również ustawić zakres, domyślnie jest to (-5, 5). Szczegóły zmiany zakresu są w komentarzu wewnątrz funkcji.

`public void setGene(int index, int value)` - ustawia pojedynczy ‘gen’ w tablicy genów osobnika. Do podanego indeksu wstawia podaną wartość.

`public int getGene(int index)` - zwraca pojedynczy ‘gen’ z tablicy genów osobnika.

Population.java

`private Individual[] individuals;` - tablica osobników dla populacji.

`public Population(int populationSize)` - konstruktor populacji, jako aargument przekazujemy liczebność populacji.

`public void initialize()` - inicjalizacja populacji, generuje losowe osobniki do tablicy osobników.

`public Individual getIndividual(int index)` - zwraca osobnika o podanym numerze indeksu z tablicy osobników.

`public Individual getFittestIndividual()` - zwraca najbardziej dopasowanego osobnika. Poprzez zmianę znaku '<' lub '>=' w pętli for możemy określić czy szukamy minimum lokalnego czy maksimum lokalnego.

`public int size()` - zwraca rozmiar tablicy osobników.

`public void saveIndividual(int index, Individual individual)` - do tablicy o podanym jako pierwszy argument indeksie zapisuje posanego w drugim argumencie osobnika.

GeneticAlgorithms.java

`private Random randomGenerator;` - deklaracja generatora liczb losowych.

`public GeneticAlgorithms()` - konstruktor algorytmu.

`public Population evolvePopulation(Population population)` - funkcja służąca do ewolucji podanej jako argument populacji.

`private void mutate(Individual individual)` - funkcja do mutacji podanego jako argument osobnika.

`private Individual crossover(Individual firstIndividual, Individual secondIndividual)` - funkcja do krzyżowania ze sobą dwóch podanych jako argumenty osobników.

`private Individual randomSelection(Population population)` - funkcja służąca do wyboru najbardziej dopasowanego osobnika. Z zadanej populacji losuje kilka osobników do tablicy o rozmiarze TOURNAMENT_SIZE z klasy Constants. Następnie zwraca najbardziej dopasowanego z nich.

Constants.java

`public static final double CROSSOVER_RATE = 0.05;` - współczynnik krzyżowania.

`public static final double MUTATION_RATE = 0.015;` - współczynnik mutacji. Najlepij, gdy jest on niewielki, rzędu 1-2%.

`public static final int TOURNAMENT_SIZE = 5;` - współczynnik służący do ustalenia liczby osobników wybranych losowo do selekcji w klasie GeneticAlgorithms.java.

`public static final int CHROMOSOME_LENGTH = 16;` - wielkość pojedynczego osobnika.

`public static final int SIMULATION_LENGTH = 1000;` - liczba symulacji.

`public static final int GENE_LENGTH = 10;` - wielkość genu.

LineChartEx.java

`public XYDataset dataset;` - dane do wykresu.

`public LineChartEx(XYDataset dataset)` - konstruktor, jako argument podaje się dane do wykresu.

`private void initUI()` - inicjalizacja i ustawienia wykresu.

`private JFreeChart createChart(XYDataset dataset)` - utworzenie wykresu z podanych danych.

App.java

`public static int option = 0;` - reprezentuje opcję, którą funkcję ma optymalizować, przekazywaną do Individual.java.

Algorytm genetyczny: output

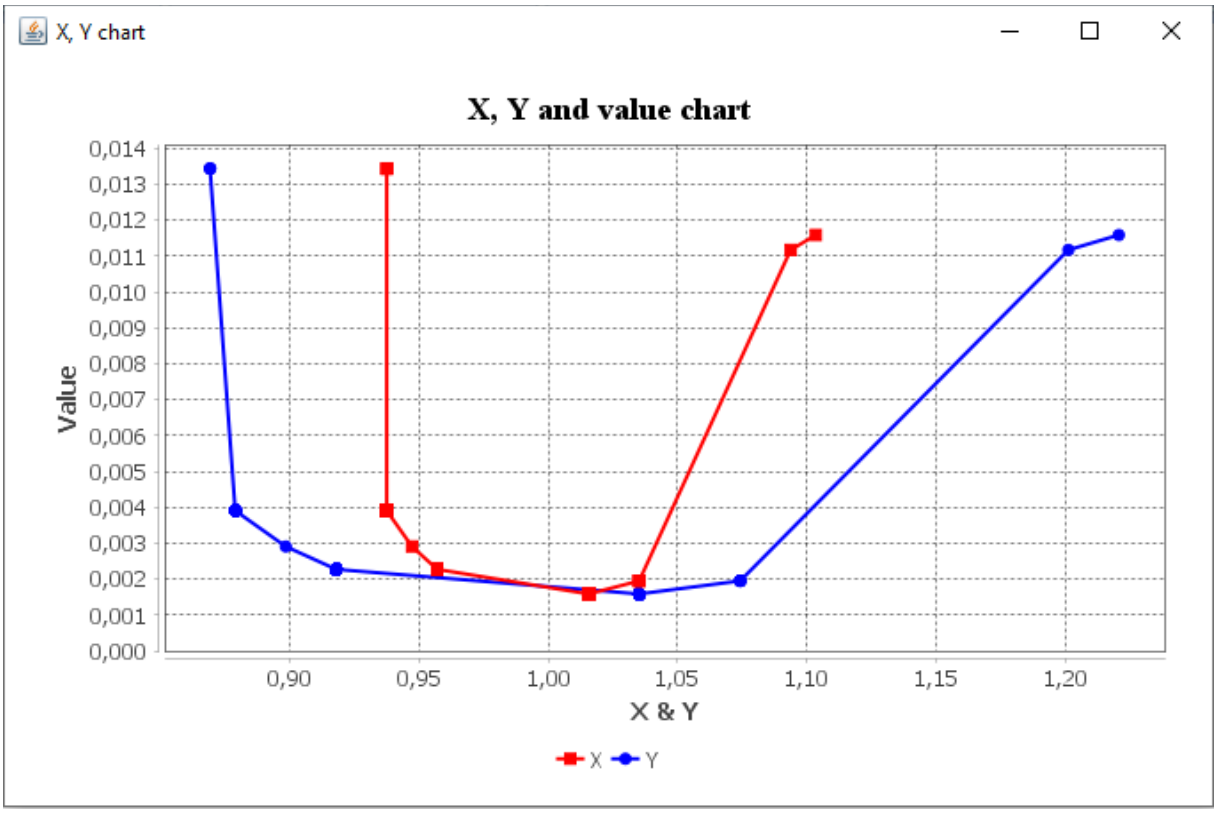
Po wyborze dostępnych opcji jako Client algorytm genetyczny rozpoczyna swoją pracę optymalizując zadany problem. Program po każdej ewolucji wyświetla numer generacji, najlepiej dopasowane minimum bądź maksimum, oraz x i y dla podanego ekstremum. Na koniec swojego działania program drukuje ostateczny najbardziej zoptymalizowany wynik i generuje wykres wartości dla współrzędnych X oraz Y.

```
Generation: 999 - minimum is: 0.001585245132446289 for:
x = 1.015625
y = 1.03515625

Generation: 1000 - minimum is: 0.001585245132446289 for:
x = 1.015625
y = 1.03515625

Solution found
x = 1.015625
y = 1.03515625
```

Rys. 2a. przykładowy output po optymalizacji funkcji Rosenbrocka



Rys. 2b. Przykładowy wykres dla funkcji Rosenbrocka. Widać wartość najbardziej dopasowanego minimum oraz jakie wartości przyjmuje X i Y.

Algorytm roju cząstek: wstęp

Ideą algorytmu PSO jest iteracyjne przeszukiwanie przestrzeni rozwiązań problemu przy pomocy roju cząstek. Każda z cząstek posiada swoją pozycję w przestrzeni rozwiązań, prędkość oraz kierunek w jakim się porusza. Ponadto zapamiętywne jest najlepsze rozwiązanie znalezione do tej pory przez każdą z cząstek (rozwiązanie lokalne), a także najlepsze rozwiązanie z całego roju (rozwiązanie globalne). Prędkość ruchu poszczególnych cząstek zależy od położenia najlepszego globalnego i lokalnego rozwiązania oraz od prędkości w poprzednich krokach. Poniżej przedstawiony jest wzór pozwalający na obliczenie prędkości danej cząstki.

$$v \leftarrow \omega v + \varphi_l r_l (l - x) + \varphi_g r_g (g - x)$$

Gdzie:

v - prędkość cząstki

ω - współczynnik bezwładności, określa wpływ prędkości w poprzednim kroku

φl - współczynnik dążenia do najlepszego lokalnego rozwiązania

φg - współczynnik dążenia do najlepszego globalnego rozwiązania

l - położenie najlepszego lokalnego rozwiązania

g - położenie najlepszego globalnego rozwiązania

x - położenie cząstki

rl, rg - losowe wartości z przedziału <0,1>

Powyższy wzór pozwala na aktualizacje prędkości wszystkich cząstek na podstawie uzyskanej do tej pory wiedzy.

Schemat działania algorytmu przedstawia się następująco:

Dla każdej cząstki ze zbioru:

- Wylosuj pozycje początkową z przestrzeni rozwiązań
- Zapisz aktualną pozycje cząstki jako najlepsze lokalne rozwiązanie
- Jeśli rozwiązanie to jest lepsze od najlepszego rozwiązanie globalnego, to zapisz je jako najlepsze
- Wylosuj prędkość początkową

Dopóki nie zostanie spełniony warunek stopu (np. minie określona liczba iteracji):
Dla każdej cząstki ze zbioru:

- Wybierz losowe wartości parametrów rl i rg
- Zaktualizuj prędkość cząstki wg powyższego wzoru
- Zaktualizuj położenie cząstki w przestrzeni
- Jeśli aktualne rozwiązanie jest lepsze od najlepszego rozwiązania lokalnego: Zapisz aktualne rozwiązanie jako najlepsze lokalnie
- Jeśli aktualne rozwiązanie jest lepsze od najlepszego rozwiązania globalnego: Zapisz aktualne rozwiązanie jako najlepsze globalnie

Źródła:

https://pl.wikipedia.org/wiki/Algorytm_genetyczny
https://www.youtube.com/channel/UCUvwlMMaeppKPdtAK8PxO8Q
https://towardsdatascience.com/how-to-define-a-fitness-function-in-a-genetic-algorithm-be572b9ea3b4
https://www.baeldung.com/java-genetic-algorithm

http://www.jfree.org/jfreechart/samples.html
http://zetcode.com/java/jfreechart
https://stackoverflow.com/questions/53734786/genetic-algorithm-java-passing-functions-with-two-coordinates?noredirect=1#comment94323283_53734786
https://en.wikipedia.org/wiki/Tournament_selection
https://www.youtube.com/watch?v=JhgDMAm-imI
http://aragorn.pb.bialystok.pl/~wkwedlo/EA6.pdf
http://www.alife.pl/optymalizacja-rojem-czastek
https://gandhim.wordpress.com/2010/04/04/particle-swarm-optimization-pso-sample-code-using-java/