Politechnika Śląska w Gliwicach

Wydział Automatyki, Elektroniki i Informatyki

Laboratorium Programowania Komputerów

Temat:

Program znajdujący minimum globalne dowolnej funkcji ciągłej za pomocą algorytmu genetycznego.

autor	Mikołaj Habarta
prowadzący	dr inż. Adam Gudyś
rok akademicki	2017/2018
kierunek	informatyka
rodzaj studiów	SSI
semestr	2
termin laboratorium	poniedziałek,10:15-11:45
grupa	1
sekcja	1
termin oddania sprawozdania	2018-06-20
data oddania sprawozdania	2018-06-20

1 Temat zadania

Tematem zadania jest poszukiwanie minimum dowolnej funkcji ciągłej za pomocą algorytmu genetycznego.

2 Analiza zadania

Zagadnienie przedstawia typowe zadanie optymalizacji funkcji, czyli znalezienie jej globalnie najmniejszej wartości.

2.1 Struktury danych

W programie wykorzystano dynamiczne tablice struktur przechowujące osobników populacji; bieżącej i następnej. Każdy osobnik na swoja wartość przystosowania, range oraz wektor genów. Takie rozwiązanie jest najoptymalniejsze, ponieważ tablica przechowywana jest w spójnym obszarze pamięci, przez co dostęp do niej jest łatwiej niż do np. listy.

2.2 Algorytmy

Do rozwiązania problemu użyto prostego algorytmu genetycznego, zaproponowanego w 1975 roku przez Johna Hollanda. Taki algorytm będzie znajdował jedynie przybliżone wartości optymalnego rozwiązania.

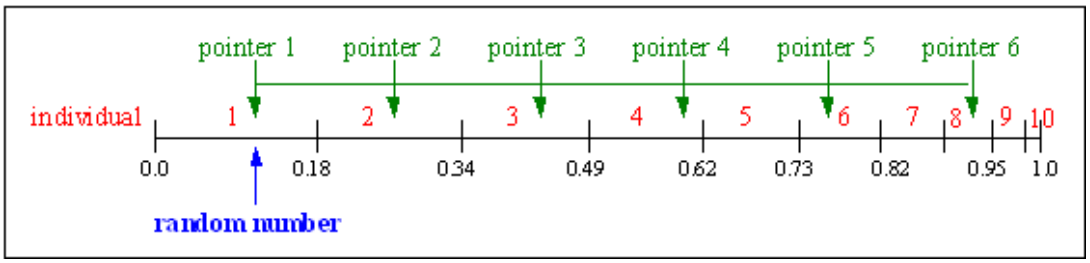
```
procedure Simple genetic algorithm
begin
t:=0
inicjacja P^0
while(not warunek stopu) do
begin
    ocena P^0
    O^t := reprodukcja P^t
    O^t:= krzyżowanie i mutacja O^t
    P^(t+1):=O^t
    t:=t+1
end
end
```

Algorytm przetwarza populację bazową (rodzicielską) $\mathbf{P^t}$ oraz populację potomną $\mathbf{O^t}$. Na początku inicjalizowana jest populacja bazowa $\mathbf{P^0}$ poprzez wypełnienie jej losowo wygenerowanymi osobnikami. Następnie algorytm przystępuje do pętli głównej programu. Następuje ocena populacji rodzicielskiej. Każdemu z osobników przypisywana jest pewna wartość wyliczona za pomocą funkcji przystosowania. W przypadku poszukiwania minimum funkcji przystosowaniem osobnika jest wartość przeciwna do wartości przeszukiwanej funkcji. Następnym krokiem jest reprodukcja, podczas której osobniki z populacji rodzicielskiej są wybierane do populacji potomnej. Prawdopodobieństwo wyboru nie jest jednak równe dla wszystkich osobników i zależy od przystosowanie osobnika. Osobniki o wyższym przystosowaniu mogą zostać wybrane nawet kilka razy. Następnie osobniki z $\mathbf{O^t}$ są poddawane operacjom genetycznym – krzyżowaniu i mutacji. Po wykonaniu tych operacji populacja potomna staje się populacją bazową. Pętla wykonuje się dopóty, dopóki nie zostanie spełniony warunek zatrzymania.

2.2.1 Metoda selekcji

Metod selekcji jest wiele i od jej wyboru zależeć może wpływ projektanta algorytmu na presję genetyczną. W tym programie wykorzystano metodę rankingową oraz próbkowanie metodą Stochastic Universal Sampling (SUS).

Metoda rankingowa polega na posortowaniu populacji niemalejąca w zależności przystosowania osobników, a następne nadaniu każdemu osobnikowi odpowiedniej rangi (najsłabszy otrzymuje rangę 1, kolejne rangę dwa itd., najlepszy osobnik w populacji o liczebności *n* otrzyma rangę *n).* Następnie prawdopodobieństwo wyboru osobnika jest liczone już tylko na podstawie jego rangi, a nie funkcji przystosowania. Każdemu z osobników jest mapowany na linii, w taki sposób, że długość odcinka należącego do danego osobnika jest równa jego randze. Następnie ustala się punkt *S/n,* gdzie *S* to suma wszystkich rangi i losowany jest punkt z przedziału (0,*p).* Osobnik, do którego należy odcinek na którym leży punkt zostaje wybrany. Następnie punkt zostaje przesunięty o *p* wybrany zostaje kolejny osobnik *(Rys. 1.0).*



Rys. 1.0

2.2.2 Kodowanie osobników

Najbardziej znanym i powszechnym jest binarne kodowanie chromosomu. Chromosom jest więc k-elementowych wektorem genów, z których każdy jest zerem lub jedynką. W programie do reprezentacji chromosomu użyjemy typu **`int64_t`,** który ma 64 bity. Aby móc osiągnąć większą dokładność chromosom będzie reprezentował liczbę w dwójkowym systemie stałoprzecinkowym.

0	*1*	*0*	*0*	*1*	*0*

Rys. 1.1 przykładowy chromosom z 6 genami

2.2.3 Operacje genetyczne

Obie operacje genetyczne – krzyżowanie i mutacja – są równie istotne dla prawidłowego działanie algorytmu. Podczas gdy pierwsza jest odpowiedzialna są eksplorację, druga umożliwia eksploatację.

Mutacja osobnika jest wykonywana dla każdego genu osobno i polega na zamianie z pewnym prawdopodobieństwem p_m wartości genu na wartość przeciwną.

Krzyżowanie – w tym przypadku używamy krzyżowania jednopunktowego – polega na rozcięciu chromosomów rodzicielskich. Następnie pierwszy fragment pierwszego chromosomu jest sklejany z drugim fragmentem drugiego chromosomu, podobnie z pierwszym fragmentem drugiego chromosomu i drugim fragmentem pierwszego. Powstałe w ten sposób łańcuchy tworzą chromosomy osobników potomnych. Wszystkie chromosomy maja taką samą długość, dlatego miejsce rozcięcia jest takie samo dla obu chromosomów i jest wybierane losowo.

2.3.4 Warunek stopu

Jako warunek końca wykonywania operacji przyjmujemy ustaloną ilość iteracji, wprowadzoną przez użytkownika.

3 Specyfikacja zewnętrzna

Program jest uruchamiany z linii poleceń. Należy przekazać do programu jako argumenty wiersza poleceń:

-liczbę generacji po przełączniku `-g`

-liczebność populacji po przełączniku `-p`

-prawdopodobieństwo mutacji po przełączniku `-d`

```
Program.exe -g 100 -p 50 -m 0.03
Program.exe -m 0.05 -g 80 -p 150
```

Przełączniki mogą być podane w dowolnej kolejności. Uruchomienie programu bez żadnych parametrów bądź z błędnymi parametrami powoduje wyświetlenie komunikatu

```
Nieprawidlowe parametry!
```

4 Specyfikacja wewnętrzna

Program został podzielony na następujące pliki:

`Header.h` – plik nagłówkowy zawierający deklaracje wszystkich funkcji oraz struktur użytych w programie

`SGA.c` – plik zawierający definicje funkcji odpowiedzialnych za wykonywanie algorytmu genetycznego.

`Source.c` plik zawierający funkcję `main`.

4.1 Typy zdefiniowane w programie

W programie zostały zdefiniowane następujące typy

```
struct osobnik {
   int64_t chromosom;
   int rank;
   double fitness;
};
```

Typ ten jest wykorzystywany do przechowywania osobnika populacji. Każdy osobnik posiada swój wektor genów (reprezentowany jako 64-bitowa liczba całkowita typu int64_t), range oraz wartość przystosowania.
W programie zdefiniowano stałą:
`#define size_64 64`
Która jest rozmiarem chromosomu i służy jako licznik w pętli w przypadku, gdy potrzebna jest iteracja po wszystkich jego elementach.

4.2 Szczegółowy opis implementacji funkcji

```
void set_base_population(struct osobnik * base, int
population);
```

Funkcja ustawia początkową populacje osobników przechowywaną w dynamicznej tablicy `base`. Dla każdego osobnika populacji funkcja losuje wartość jego chromosomu.

```
void shuffle(struct osobnik* p, int population);
```

Funkcja tasuje tablicę `p` o rozmiarze `population`, losowo zmieniając jej elementy.

```
void choose_and_cross(struct osobnik *p,int population);
```

Funkcja za pomocą funkcji `shuffle` tasuje tablicę `p`, a następnie dla każdych dwóch kolejnych elementów tablicy wywołuje losuje pewną liczbę całkowitą z przedziału `(0,size_64)` a następnie wywołuje funkcję `crossover` na chromosomach tych elementów oraz na wylosowanej liczbie.

```
void crossover(int64_t* p1, int64_t* p2, int cross_point);
```

Funkcja dokonuje krzyżowania chromosomów `p1` i `p2`. Wartości tych chromosomów są modyfikowane wewnątrz funkcji. Krzyżowanie polega na zamianie ze sobą pewnej liczby mniej znaczących bitów . Liczba ta jest przekazywana do funkcji jako parametr `cross_point`.

```
void judge_population(struct osobnik *pop,int population);
```

Funkcja dokonuje oceny przystosowania osobników danej populacji i zapisuje ją w zmiennej `fitness`. Aby tego dokonać wywołuje funkcję `fitness_function` chromosomów osobników populacji `pop`.

```
double fitness_function(int64_t chrom);
```

Funkcja konwertuje liczbę stałoprzecinkową reprezentowaną przez chromosom `chrom` na liczbę zmiennoprzecinkową typu `double`, a następnie wywołuje zadaną funkcję ciągłą na tej liczbie.

```
void choose_population(struct osobnik* parents, struct osobnik* offspring, int population);
```

Funkcja losuje nowe osobniki z populacji `parents` do populacji `offspring`. Prawdopodobieństwo wylosowania jest większe dla osobników o większej wartości funkcji przystosowania.

```
void sort(struct osobnik *p, int left, int right);
```

Funkcja sortuje niemalejąco tablicę `p`. Sortowanie jest rekurencyjnie i odbywa się za pomocą algorytmu sortowania szybkiego.

```
void mutate(struct osobnik* m,double probability, int population);
```

Funkcja z pewnym zadanym prawdopodobieństwem rekurencyjnie mutuje każdy gen na chromosomie wszystkich osobników populacji `m`. Mutacja oznacza zanegowanie danego bitu.

```
void copy(struct osobnik* parents, struct osobnik* offspring, int population);
```

Funkcja kopiuje wszystkie elementy z tablicy `offspring` do tablicy `parents`. `population` jest rozmiarem tych tablic.

```
double function_1(double x);
```

Przykładowa funkcja ciągła, która została wykorzystana w celach testujących.

5 Testowanie

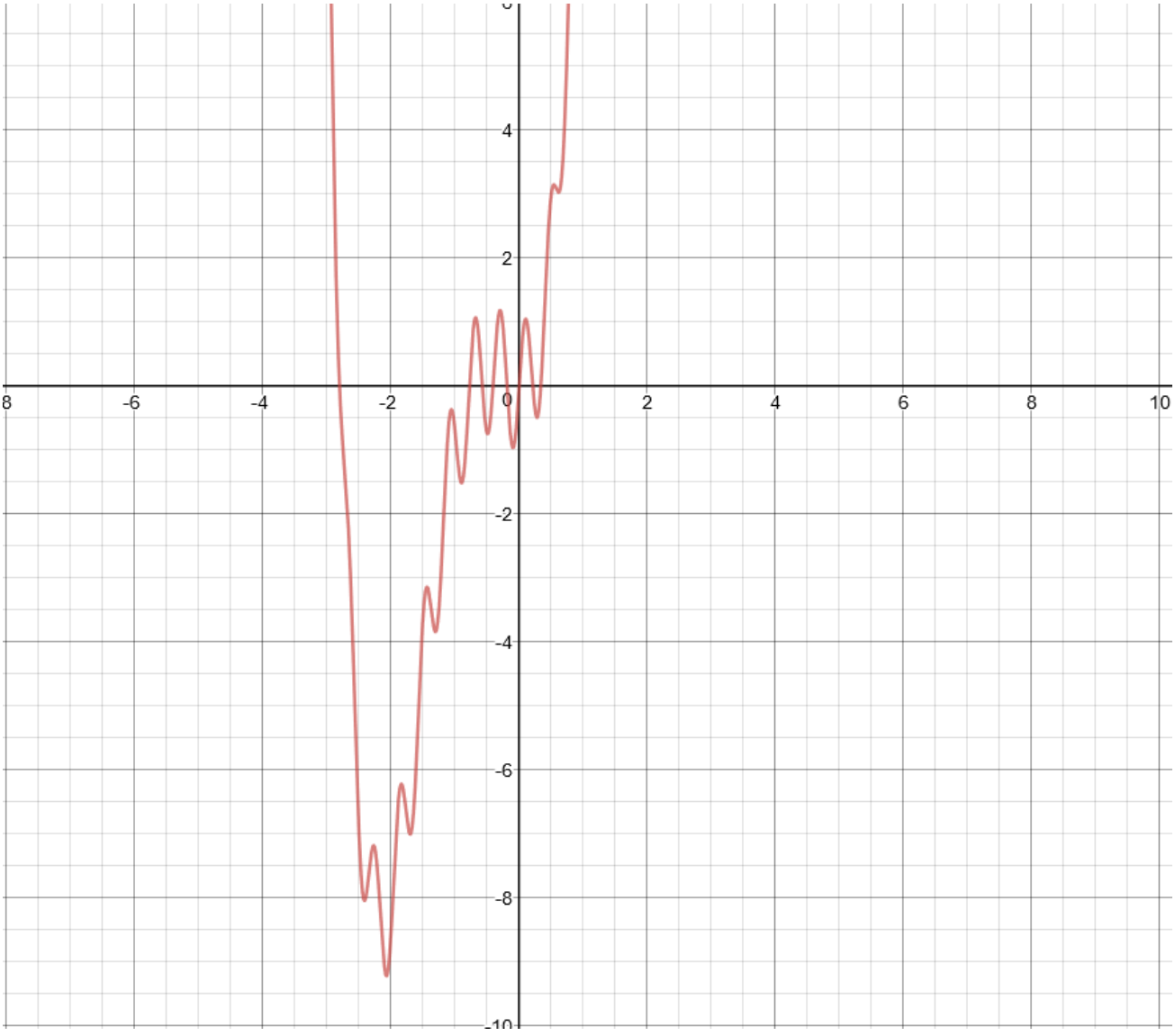
Testowanie programu odbyło się za pomocą funkcji ciągłej.

$f(x) = 2x^4 + 6\ x^3 + x^3 + 4\ x^2 + \sin(16x)$

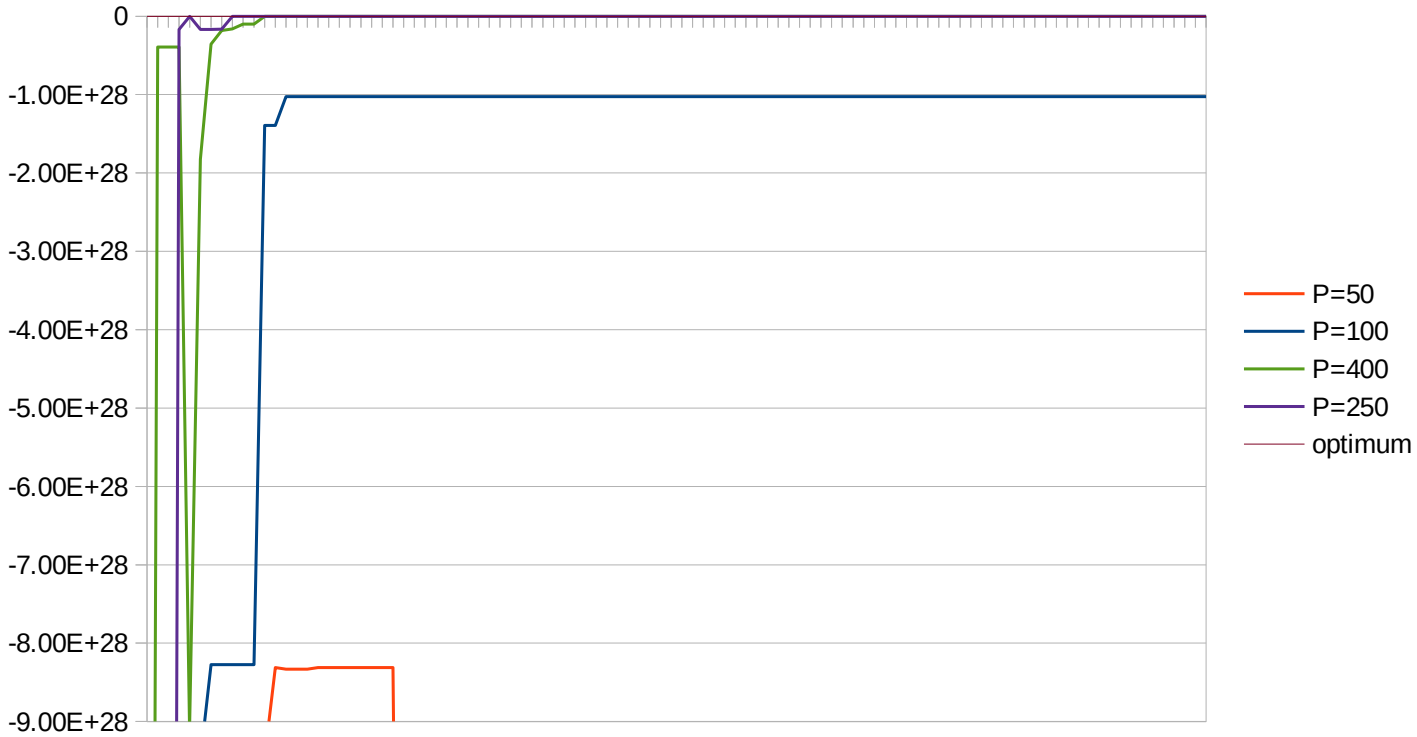
Ponieważ funkcja posiada wiele minimów lokalnych oraz zmienia się gwałtownie na stosunkowo krótkim przedziale, może stanowić wyzwanie dla programu.

Globalne minimum tej funkcji wynosi -9.2192, a więc największa możliwa wartość funkcji przystosowania wynosi 9.2192.

Wykres funkcji wygląda w następujący sposób:

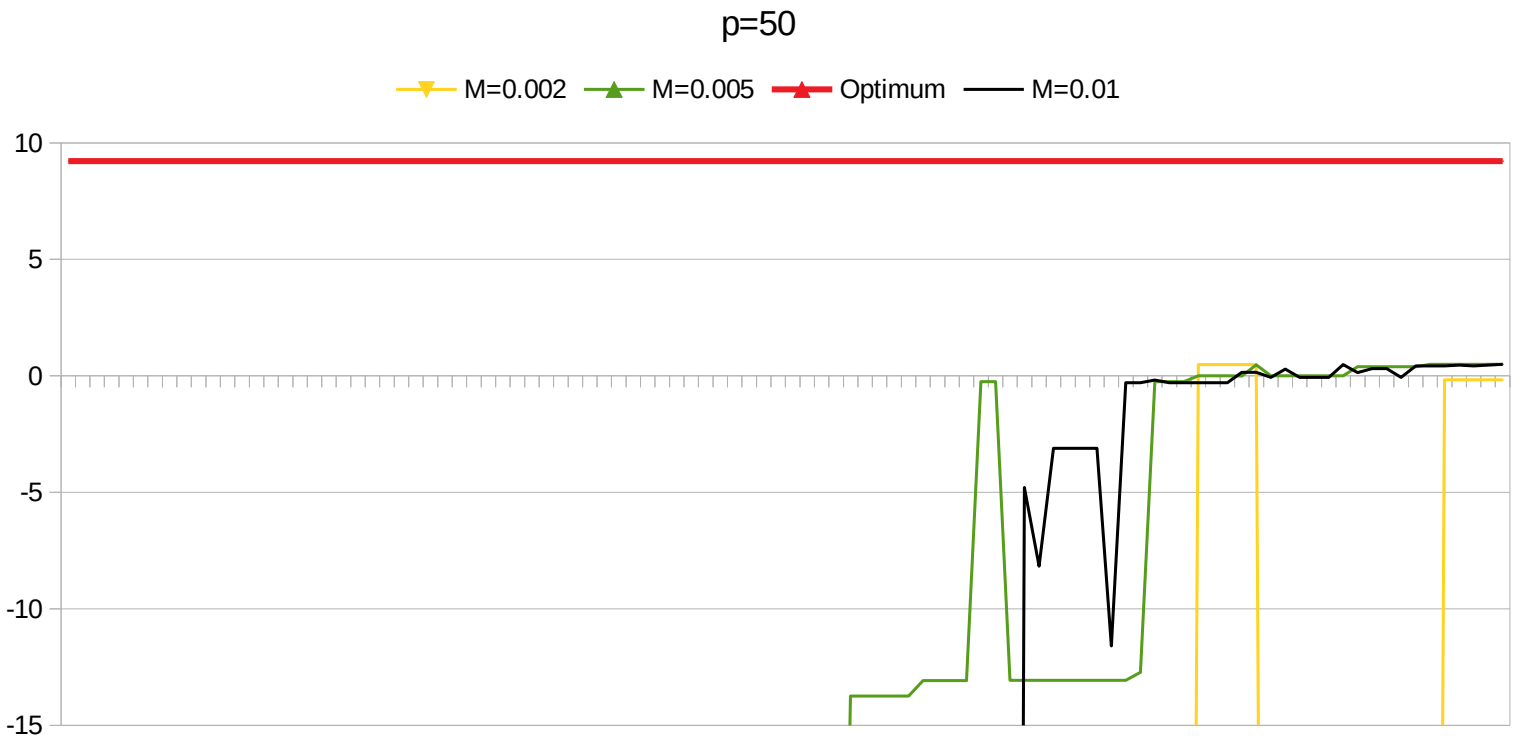


Dla prawdopodobieństwa mutacji M=0 zbadano zależność liczebności populacji na osiągnięty wynik.

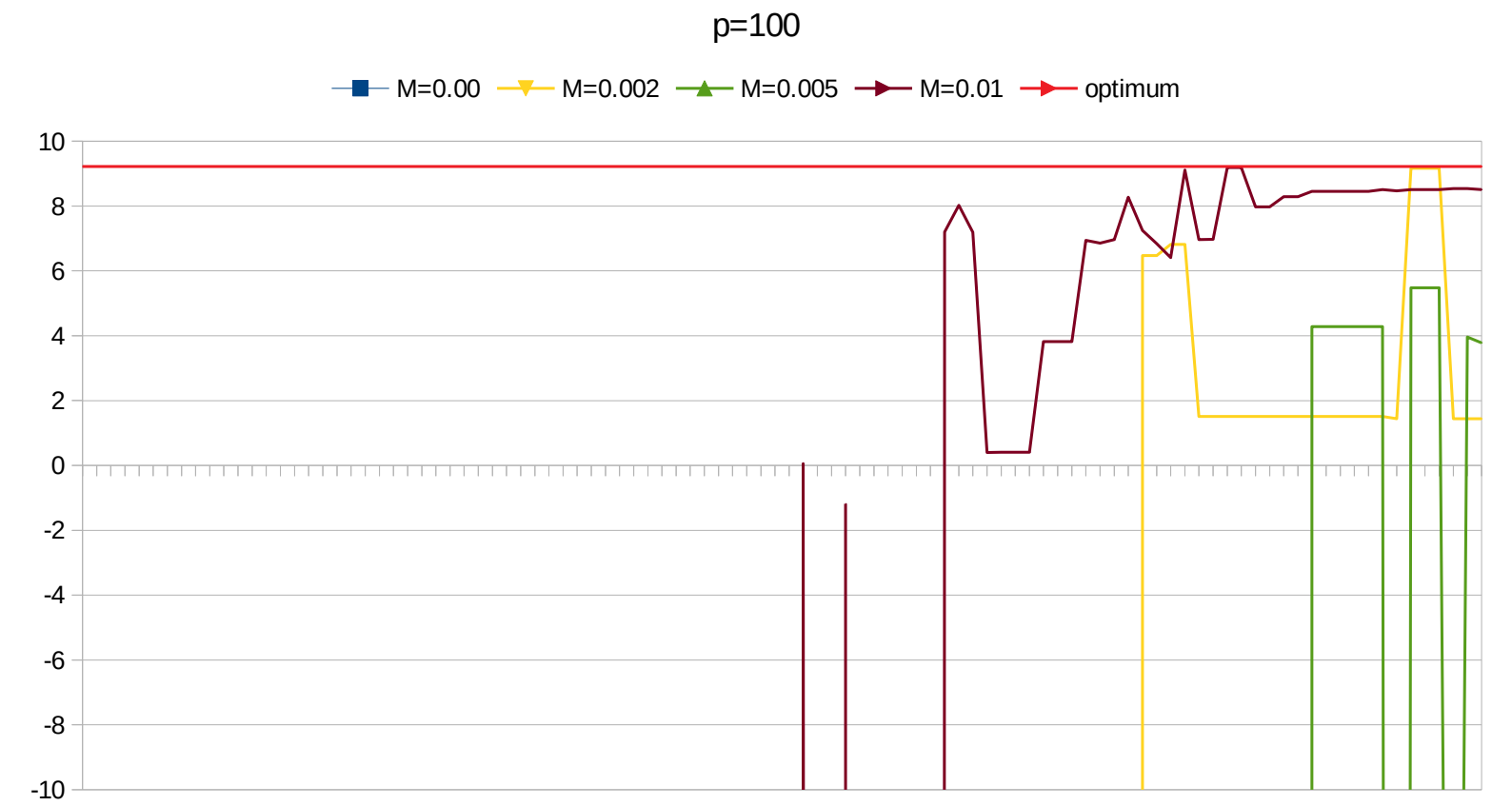


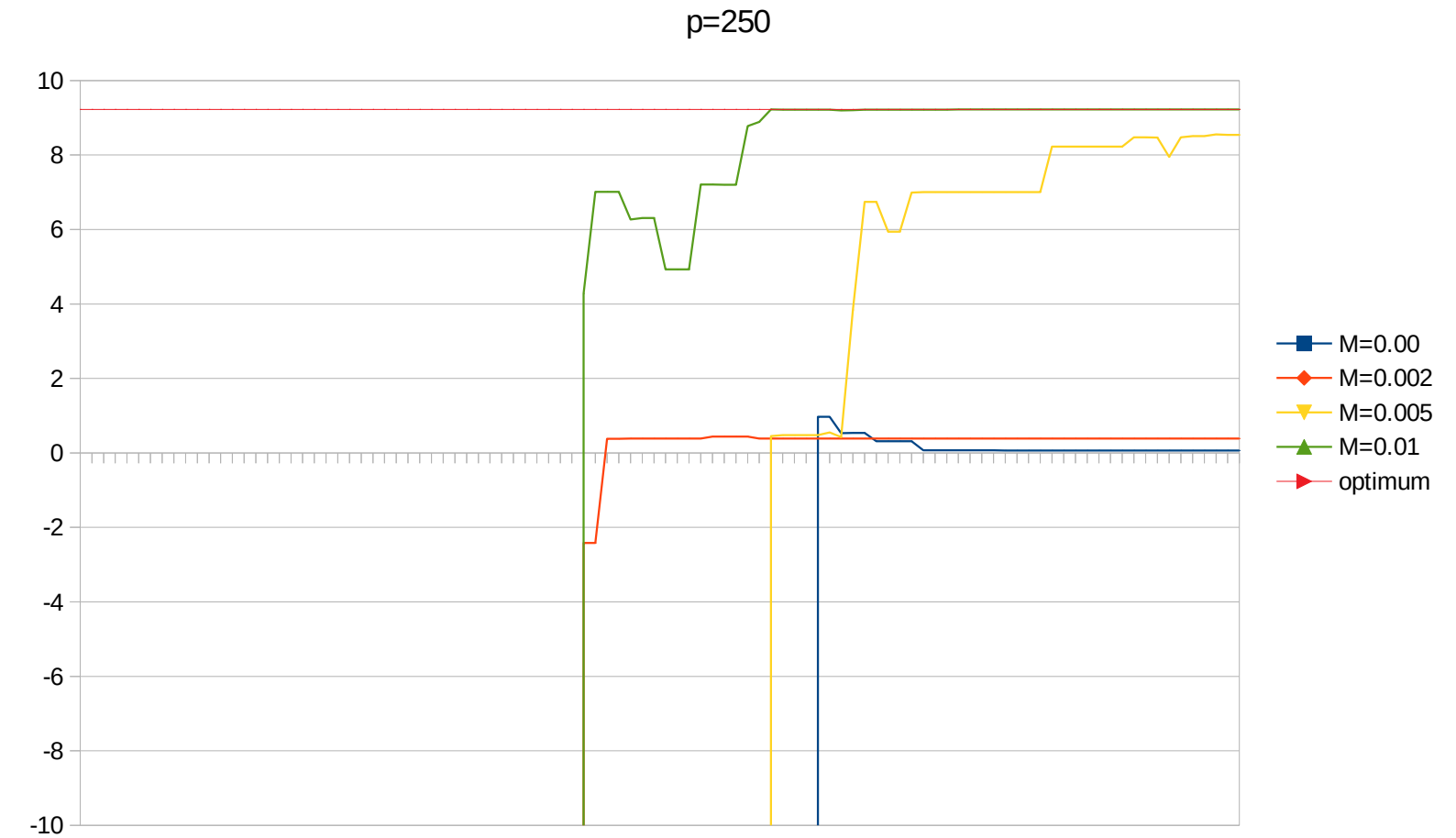
Jak widać brak mutacji doprowadził do braku różnorodności genetycznej w przypadku niskich liczby populacji. Dla większej liczby członków populacji udało się wyjść z tej stagnacji genetycznej, jednak nie zostało osiągnięte globalne minimum, a jedynie lokalne.

Następnie dla stałej liczby członków populacji zmieniano prawdopodobieństwo mutacji:

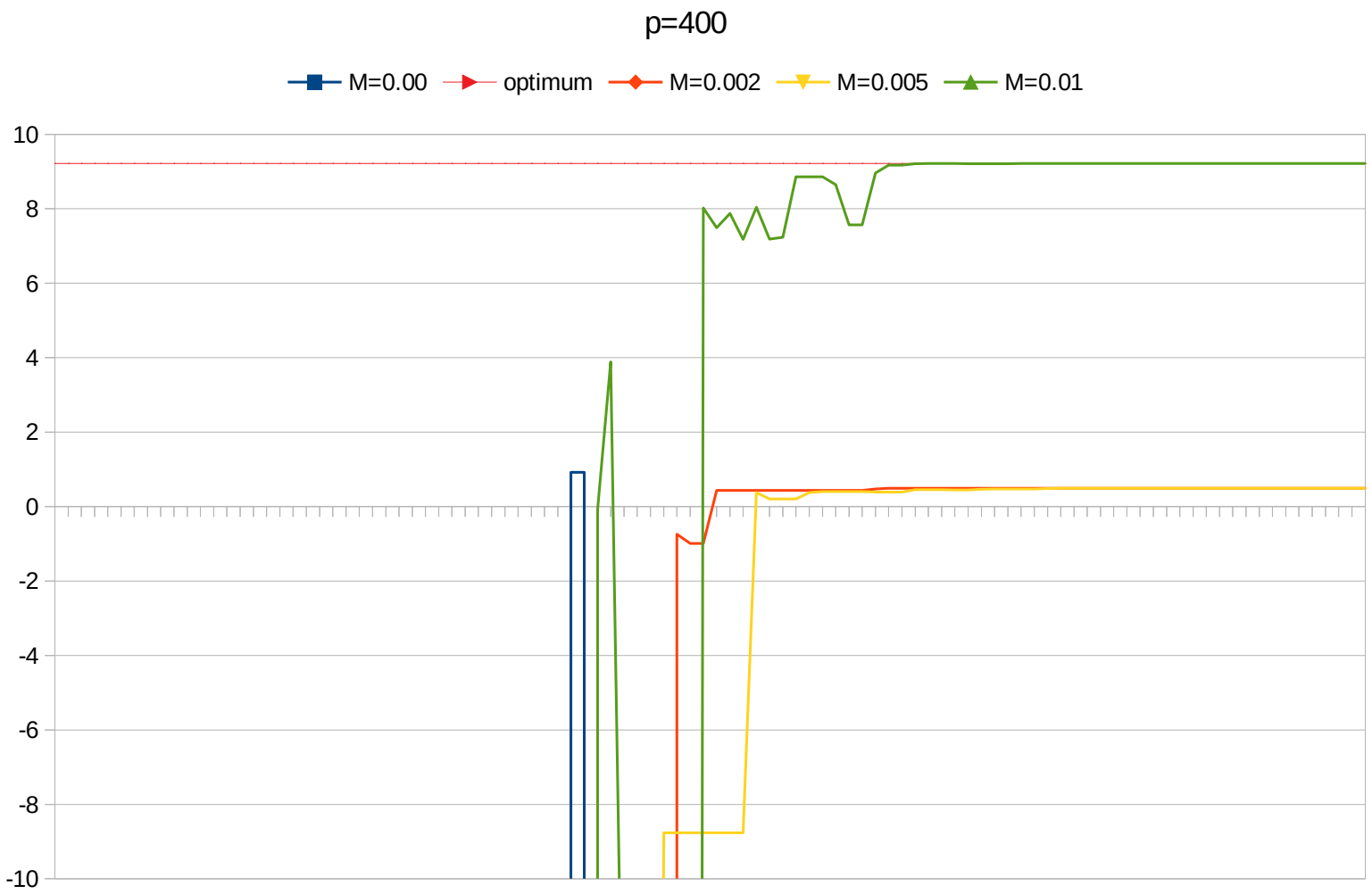


Najlepszy wynik jaki udało się uzyskać to 0,49, co jest jedynie minimum lokalnym funkcji.





Dla prawdopodobieństwa mutacji = 0.01 udało się osiągnąć wynik 9.219194, czyli z dość dużą dokładnośią znaleźć minimum globalne.



W tym przypadku również prawdopodobieństwo mutacji M=0.01 dało najlepszy rezultat.

6 Wnioski

Udało się z powodzeniem napisać program, który przy pomocy prostego algorytmu genetycznego wyznacza z pewnym przybliżeniem globalne minimum dowolnej funkcji ciągłej. Dokładność wyznaczonego przybliżenia zależy od początkowej liczby populacji, prawdopodobieństwa mutacji oraz od czynnika losowego.