Politechnika Śląska w Gliwicach
Wydział Automatyki, Elektroniki i Informatyki

Programowanie Komputerów 3

Algorytm Genetyczny

autor	Marek Hermansa
prowadzący	dr inż. Piotr Pecka
rok akademicki	2017/2018
kierunek	informatyka
rodzaj studiów	SSI
semestr	3
termin laboratorium / ćwiczeń	czwartek, 08:00 – 9:30
grupa	2
termin oddania sprawozdania	2018-02-04
data oddania sprawozdania	2018-02-03

1 Treść zadania

Napisać nastepujący program. Prosty algorytm genetyczny składający się z podstawowych operatorów mutacji, krzyżowania, selekcji (można wybrać dowolne metody). Funkcja celu powinna reprezentować wybrany problem. W każdej iteracji na konsoli powinno wypisać się numer generacji, wartość dopasowania najlepszego osobnika.

2 Analiza zadania

Zagadnienie przedstawia problem znajdowania globalnego minimum ustalonej funkcji (tu: McCormick[1]) dwu zmiennych za pomocą algorytmu genetycznego w ustalonym przedziale. W każdej iteracji na konsoli wypisywany jest numer generacji, wartość zmiennej x, wartość zmiennej y i odpowiadająca tym wartościom wartość funkcji będąca najlepszym rozwiązaniem problemu w danej populacji.

2.1 Struktury danych

W programie wykorzystano strukturę `vector`, do reprezentacji populacji (wewnątrz klasy, jako pole). Elementami struktury są obiekty klasy `Chromosome`. Taka struktura danych umożliwia łatwe dodawanie kolejnych chromosomów przy generacji populacji. Ponadto zmiana liczebności populacji wymaga zmiany wartości tylko jednej zmiennej - `vector` jest strukturą dynamiczną.

2.2 Algorytmy

Mechanizm działania algorytmu genetycznego polega na kopiowaniu ciągów i wymianie podciągów; składa się z nastepujących kroków:

1. Inicjacja – utworzenie populacji początkowej, poprzez losowy wybór ustalonej liczby chromosomów.

2. Ocena przystosowania – obliczenie wartości funkcji przystosowania dla każdego chromosomu.

3. Selekcja chromosomów – wybór chromosomów, które biorą udział w tworzeniu nowej populacji.

4. Zastosowanie operatorów genetycznych – na grupie chromosomów, wybranej drogą selekcji działają operatory genetyczne: krzyżowania i mutacji;

[1] `https://www.sfu.ca/~ssurjano/mccorm.html`

Operacja krzyżowania polega na wymianie fragmentów łańcuchów dwóch chromosomów rodzicielskich. Krzyżowanie jest kluczowym operatorem w algorytmach genetycznych, stanowiącym o ich sile i efektywności. krzyżowanie zachodzi znacznie częściej niż mutacja.

Mutacja odgrywa znacznie mniejsza rolę. Mutacja polega na wprowadzeniu do istniejących zakodowanych chromosomów, pewnych losowych zmian.

Selekcja turniejowa polega na podzieleniu populacji na podgrupy k-elementowe (tu: 2) i wyborze z każdej podgrupy osobnika o najlepszym przystosowaniu.

5. Utworzenie nowej populacji – Chromosomy otrzymane jako rezultat działania operatorów genetycznych na chromosomy tymczasowej populacji wchodzą w skład nowej populacji.

6. Wyprowadzenie najlepszego chromosomu – najlepszym rozwiązaniem jest chromosom o najmniejszej (szukane jest minimum) wartości funkcji przystosowania.

Nową populację tworzą chromosomy powstałe w wyniku selekcji i działania operatorów genetycznych. Chromosomy są zakodowanie binarnie (standard IEEE 754) - geny mogą przyjmować tylko wartości 0 i 1. Selekcji dokonuje się z wykorzystaniem metody turnieju. Krzyżowanie jest jednopunktowe.

3 Specyfikacja wewnętrzna

Program został zrealizowany zgodnie z paradygmatem obiektowym.

3.1 Typy zdefiniowane w programie

W programie zdefiniowano nastepujące klasy:

- `GeneticAlgorithm`
- `Population`
- `Chromosome`
- `Record`

Poniżej znajduje się opis każdej klasy wraz z przedstawieniem deklaracji ich metod i pól.

3.1.1 GeneticAlgorithm

Klasa `GeneticAlgorithm` uruchamia algorytm genetyczny, stosuje go do populacji chromosomów z intencją stopniowego zwiększania ogólnego przystosowania w kolejnych iteracjach.

```
class GeneticAlgorithm
{
public:

GeneticAlgorithm(void);
~GeneticAlgorithm(void);

void Load( const int& cratio, const int& mratio,
 const int& psize, const int& iter,
 const int& csize, const int& ssize,
 const std::string& path);
void Start();

private:

void CreatePopulation();
double Evaluate();

void Crossover();
void Mutate();
void Select();

void SetParameters(const int& cratio, const int& mratio,
const int& psize, const int& iter, const int& csize,
const int& ssize );

void RecordResult(const double& result, const int& iter);

private:

int numberGenerations;

int mutationRatio;
int crossoverRatio;
```

```
int populationSize;
int chromosomeSize;
int selection_size;

double bestFitness;
int bestFitnessIndex;
float best_x;
float best_y;

Population pop;
Record record;
};
```

3.1.2 Population

Klasa `Population` przechowuje populację obiektów `Chomosome` oraz stosuje do nich operatory `Crossover`, `Mutate` i `Select`; generuje początkową populację obiektów `Chromosome` z losowymi wartościami; wykorzystuje inne funkcje do konwersji pomiędzy binarnymi i dziesiętnymi reprezentacjami ciągów bitów.

```
class Population
{
public:

Population(void);
~Population(void);

void CreateRandomPopulation(const int& size);
double EvaluatePopulation(float& bx, float& by);

void Mutation(const int& index, const int& mutationRatio);
void Crossover(const int& index1, const int& index2, const int& point);

void SetChromosomeSize(const int& size);
double CalcChromosomeFitness(const int& index, float& xv, float& yv);
double GetChromosomeFitness(const int& index) const;
void CopyChromosome(const int& source, const int& dest);

private:

Chromosome* CreateRandomChromosome();
```

```
std::string GetXstring(Chromosome* chr);
std::string GetYstring(Chromosome* chr);

float GetFloat32_IEEE754(std::string Binary);
int Binary32ToHex(std::string Binary);

double CalculateFitnessFunction(const float& x, const float& y);

private:

std::vector<Chromosome*> pop;
int chrSize;
};
```

3.1.3 Chromosome

Klasa `Chromosome` przechowuje ciąg bitów, który reprezentuje wartości binarne; wykorzystuje funkcje do wykonywania elementarnych operacji na ciągach.

```
class Chromosome
{
public:

Chromosome(const int& size);
~Chromosome(void);

void SetChromosome( const int& index, const unsigned char& value );
unsigned char GetChromosome( const int& index );

void SetFitness( const double& value );
double GetFitness() const;

int GetSize() const;

private:

std::vector<int> chr;

int chrSize;
double fitness;
};
```

3.1.4 Record

Klasa `Record` obsługuje zapis statystyk do pliku oraz drukowanie ich w konsoli.

```
class Record
{
public:

Record();
Record(char*);
~Record();

void Open(const char*);
void Write(char* txt);

private:

ofstream m_stream;
};
```

3.2 Ogólna struktura programu

W funkcji głównej wywoływana jest funkcja

```
genetic_algorithm.Load(crossover_ratio, mutation_ratio,
population_size, number_generations, chromosome_size,
selection_size, path);
```

odpowiedzialna za przekazanie wartości parametrów z funkcji głównej do obiektu. Następnie wywoływana jest funkcja

```
genetic_algorithm.Start();
```

odpowiedzialna za uruchomienie algorytmu.

3.3 Szczegółowy opis implementacji funkcji

Poniżej znajduje się opis wszystkich najważniejszych metod każdej z klas.

3.3.1 GeneticAlgorithm

```
void Load( const int& cratio, const int& mratio, const int& psize,
const int& iter, const int& csize, const int& ssize,
const std::string& path);
```

Funkcja inicjalizuje wartości parametrów wywołując funkcję `SetParameters(cratio, mratio, psize, gener, csize, ssize)`; generuje populację poczatkową wywołując funkcję `CreatePopulation()`; przygotowuje do zapisu i wyświetlania statystyk wywołując funkcję `Record.Open(path.c_str())`.

```
void Start();
```

Funkcja wywołuje w pętli funkcje: `Evaluate()`, która dokonuje oceny populacji pod względem przystosowania; `RecordResult(Evaluate(), i)`, która jest odpowiedzialna na zapis i wyświetlanie wyników; `Select()`, która realizuje operator selekcji; `Crossover()`, która realizuje operator krzyżowania; `Mutate()`, która realizuje operator mutacji. Ilość przejść pętli jest równa zmiennej `numberIterations`, która reprezentuje pożądaną liczbę pokoleń.

```
void CreatePopulation();
```

Funkcja wywołuje metodę `CreateRandomPopulation(populationSize)` klasy `Population`, na obiekcie `pop`, która jest odpowiedzialna za generację losowej populacji.

```
double Evaluate();
```

Funkcja określa domyślne wartości zmiennych `bx`, `by`, które reprezentują x i y chromosomu o najlepszym[2] przystosowaniu; wywołuje funkcję `EvaluatePopulation(bx, by)`, której wartość zwrócona to najlepsze przystosowanie w danej populacji; jeśli zwrócona wartość jest lepsza niż wartość najlepszego przystosowania w poprzedniej populacji, wartość poprzednia jest nadpisywana nową wartością.

```
void Crossover();
```

[2]Używane jest sformułowanie lepsze/gorsze przystosowanie, a nie mniejsza/większa wartość przystosowania, ponieważ zadaniem algorytmu jest poszukiwanie minumum funkcji, stąd im mniejsza wartość przystosowania tym lepiej.

Funkcja dokonuje wyboru dwóch chromosomów, które zostaną poddane krzyżowaniu oraz wyznacza punkt przecięcia; wywołuje metodę `Crossover(populationSize)` klasy `Population`, na obiekcie pop, która dokonuje właściwej operacji krzyżowania na wybranych chromosomach. O liczbie wywołań decyduje współczynnik `crossoverRatio`.

```
void Mutate();
```

Funkcja wywołuje metodę `Mutation(i, mutationRatio)` klasy `Population`, na obiekcie `pop`, która dokonuje właściwej operacji mutowania na wybranych chromosomach. O liczbie wywołań decyduje współczynnik `mutationRatio`.

```
void Select();
```

Funkcja z bieżącej populacji osobników wybiera pary chromosomów; nastepnie zachowuje ten, którego przystosowanie jest większe. O liczbie szukanych par decyduje współczynnik `selectionSize`. Wybór dokonuje się metodą turniejową.

```
void SetParameters(const int& cratio, const int& mratio,
const int& psize, const int& iter, const int& csize,
const int& ssize );
```

Funkcja przypisuje odpowiednie wartości parametrom odpowiedzialnym za działanie operatorów, liczbę pokoleń oraz rozmiar chromosomów i populacji.

```
void RecordResult(const double& result, const int& iter);
```

Funkcja zapisuje najlepszy wynik każdej populacji do pliku oraz wyświetla go w konsoli. Ścieżka pliku jest określana w funkcji głównej.

3.3.2 Population

```
void SetChromosomeSize(const int& size);
```

Funkcja przypisuje wartość zmiennej `chrSize`, która określa długość chromosomu.

```
void CreateRandomPopulation(const int& size);
```

Funkcja wywołuje w pętli funkcję `CreateRandomChromosome()`, tworząc losową populację chromosomów. O liczbie wywołań decyduje współczynnik `size`.

```
void Crossover(const int& index1, const int& index2, const int& point1);
```

Funkcja zamienia części ustalonej pary chromosomów wyznaczone przez określony wcześniej punkt. Zamiana odbywa się przez wymianę pojedynczych bitów.

```
void Mutation(const int& index, const int& mutationRatio);
```

Funkcja dokonuje zmiany losowych bitów chromosomu. Zmiana jest dokonywana w części chromosomu, który reprezentuje x albo y, nigdy w obu jednocześnie. O liczbie zmienionych bitów decyduje współczynnik `mutationRatio`. Wartość bitu nie jest zamieniana na przeciwny, ale losowana jest nowa wartość (0 albo 1).

```
double EvaluatePopulation(float& bx, float& by);
```

Funkcja wywołuje w pętli funkcję `CalcChromosomeFitness(i, x, y)`, która zwraca wartość przystosowania danego chromosomu oraz wartości x i y (przez referencję). Jeśli zwrócona wartość przystosowania jest lepsza niż dotychczasowa najlepsza wartość przystosowania w populacji, jest nadpisywana; nadpisywane są również wartości x i y.

```
double CalcChromosomeFitness(const int& index, float& xv, float& yv);
```

Funkcja kolejno wywołuje funkcję `GetXstring(chr)` uzyskując ciąg bitów reprezentujący wartość zmiennej x; wywołuje funkcję `GetFloat32_IEEE754(xstr)` konwertując uzyskany ciąg na liczbę dziesiętną; wywołuje funkcję `GetYstring(chr)` uzyskując ciąg bitów reprezentujący wartość zmiennej y; wywołuje funkcję `GetFloat32_IEEE754(ystr)` konwertując uzyskany ciąg na liczbę dziesiętną; wywołuje funkcję `CalculateFitnessFunction(x, y)` uzyskując wartośc przystosowania danego chromosomu.

```
double GetChromosomeFitness(const int& index) const;
```

Funkcja zwraca wartość przystosowania danego chromosomu.

```
void CopyChromosome(const int& source, const int& dest);
```

Funkcja nadpisuje zawartość jednego chromosomu (`dest`) zawartością drugiego (`source`).

```
Chromosome* CreateRandomChromosome();
```

Funkcja generuje losowo ciąg zer i jedynek, który reprezentuje chromosom.

```
std::string GetXstring(Chromosome* chr);
```

Funkcja wyłuskuje z ciągu, który reprezentuje chromosom, część reprezentującą wartość zmiennej x.

```
std::string GetYstring(Chromosome* chr);
```

Funkcja wyłuskuje z ciągu, który reprezentuje chromosom, część reprezentującą wartość zmiennej y.

```
float GetFloat32_IEEE754(std::string Binary);
```

Funkcja dokonuje konwersji liczby zapisanej w formacie IEEE-754 na liczbę dziesiętną. Liczbę konwertowana w formacie IEEE-754 zapisana jest za pomocą trzydziestu dwóch bitów. Pierwszym bitem jest bit znaku. Jeśli liczba jest ujemna bit znaku ma wartość 1; Jeśli liczba jest dodatnia bit znaku ma wartość 0. Dalej jest 8 bitów kodujących cechę (zakres [-127, 128]). Kolejne 23 bity to mantysa liczby.

```
int Binary32ToHex(std::string Binary);
```

Funkcja konwertuje 32-bitowy ciąg na liczbę szesnastkową (na potrzeby funkcji `GetFloat32_IEEE754(std::string Binary)`).

```
double CalculateFitnessFunction(const float& x, const float& y);
```

Funkcja zwraca wartość danej funkcji dwu zmiennych na podstawie przekazanych wartości x i y. Jeśli wartość zmienej x lub y wykracza poza rozpatrywany obszar, przystosowanie danego chromosomu jest pogarszane; powoduje to skupienie na rozpatrywanym obszarze.

3.3.3 Chromosome

```
void SetChromosome(const int& index, const unsigned char& value);
```

Funkcja ustawia wartość określonego bitu chromosomu.

```
unsigned char GetChromosome(const int& index);
```

Funkcja zrwaca wartość określonego bitu chromosomu.

```
void SetFitness(const double& value);
```

Funkcja ustawia wartość przystosowania.

```
double GetFitness() const;
```

Funkcja zwraca wartość przystosowania.

```
int GetSize() const;
```

Funkcja zwraca długość chromosomu.

4 Testowanie

Przed algorytmem postawiono zadanie znalezienia globalnego minimum funkcji McCormick[3] (Rysunek 1). Funkcja jest określona wzorem:

$$f(x, y) = \sin(x + y) + (x - y)^2 - 1.5x + 2.5y + 1$$

Globalne minimum funkcji wynosi w przybliżeniu:

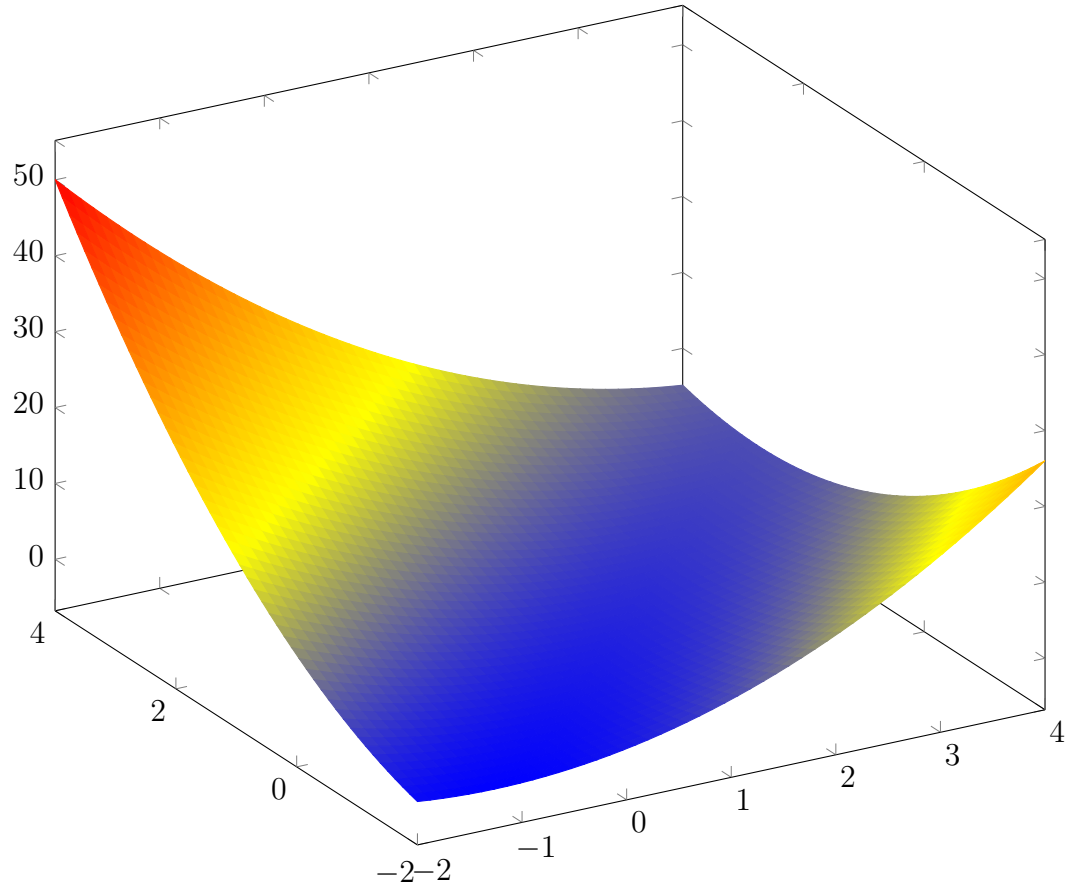
$$f(-0.54719756, -1.54719755) = -1.91322295$$

Obszar poszukiwań minimum został ograniczony do:

$$-1.5 \leq x \leq 4.0$$

$$-3.0 \leq y \leq 4.0$$

[3] https://www.sfu.ca/~ssurjano/mccorm.html



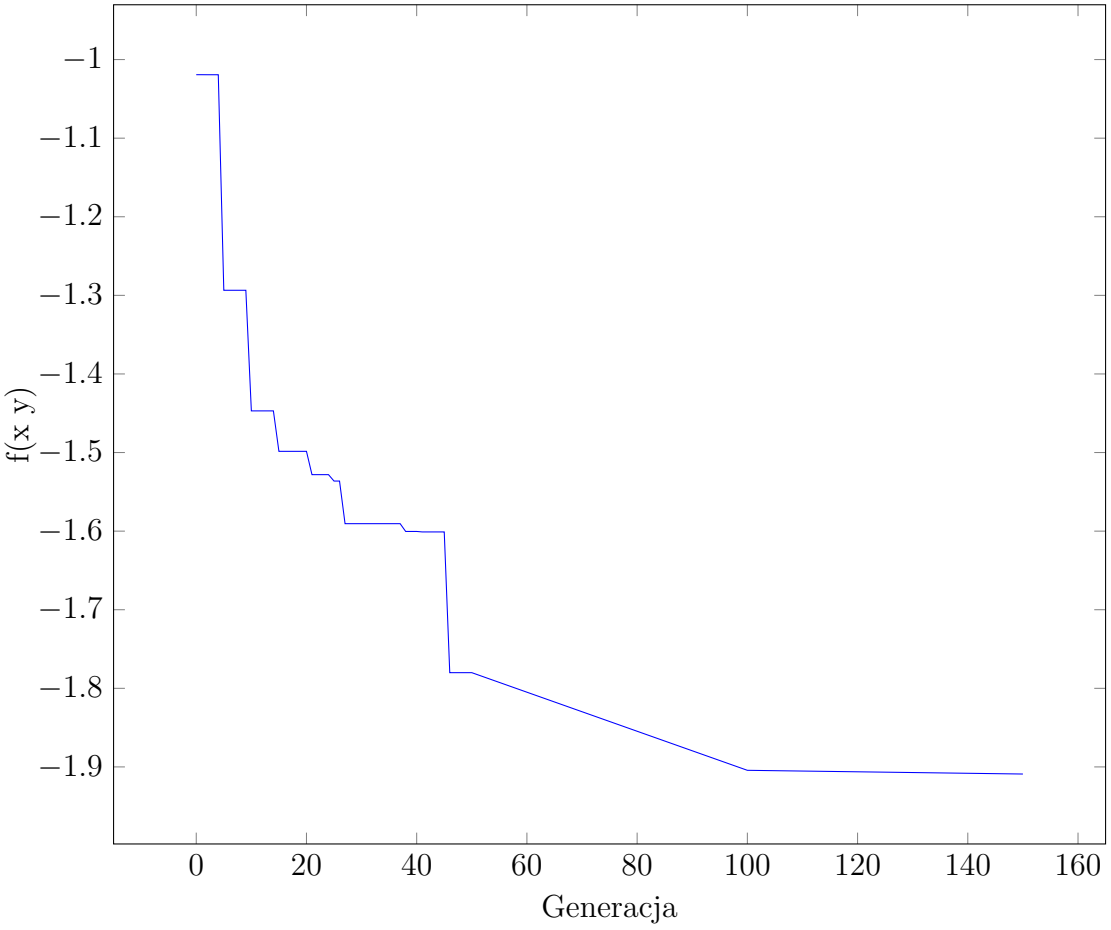
Rysunek 1: Funkcja McCormick

Zadanie ustalenia najbardziej optymalnych wartości parametrów `population_size`, `crossover_ratio`, `mutation_ratio`, `selection_size` nie jest zadaniem łatwym. Jednym z rozwiązań tego problemu jest wykorzystanie algorytmu genetycznego do ustalenia wartości parameterów, które gwarantują najskuteczniejsze działanie. To jednak wykracza poza temat projektu. Stąd ustalono wartości parametrów metodą prób i błędów, wykorzystując znajomość szczegółów implementacji oraz teorii. Ustalono, że poniższe wartości parametrów dają zadowalające rezultaty.

- `population_size = 500`,
- `crossover_ratio = 50`,
- `mutation_ratio = 20`,
- `selection_size = population_size / 10 = 50`

W tabelach 1, 2 i 3 na stronach, odpowiednio 17, 18, 19 przedstawiono rezultaty działania algorytmu w omówionych warunkach. Na Rysunku 2 przedstawiono zmianę wartości funkcji $f(x, y)$ w kolejnych generacjach w omówionych warunkach. Po 150 generacji, zmiana jest niezauważalna w użytej skali.

Około generacji 2150 funkcja przyjmuje wartość -1.91322, która nie ulega zmianie aż do ostatniego wygenerowanego pokolenia. Wartość ta jest zgodna z oczekiwaną. Precyzję można zwiększyć przez wykorzystanie innego sposobu kodowania. W obecnym sposobie kodowania mantysę reprezentują 23 bity, co daje 6-cyfrową mantysę w postaci dziesiętnej. Program uruchomiono również ustawiając większą precyzję wyświetlanych wartości - uzyskany wynik był dokładniejszy od prezentowanego o 1 cyfrę po przecinku (-1.913222).



Rysunek 2: Zmiana wartości funkcji f(x,y) w kolejnych generacjach

5 Wnioski

- Algorytm genetyczny jest przykładem algorytmu, którego implementacja w paradygmacie obiektowym jest intuicyjna. Wybór obiektów, dla których warto stworzyć klasy jest oczywisty, szczególnie przy prostej implementacji.

- Implementacja algorytmu genetycznego nie sprawia dużych problemów - w podstawowej wersji jest to prosty algorytm, który można rozbudować dodając kolejne operatory i zwiększając ich złożoność.

- Testowanie algorymtu często trwało kilka minut, co wydłużało czas pracy. Testy przeprowadzano najczęściej na populacji o liczebności 500 i dla 1000 iteracji. Zmniejszenie liczebności populacji znacznie przyśpiesza proces testowania, ale często zaburza rezultaty. Innym sposobem na przyśpieszenia działania programu mogłoby być stworzenie w obrębie programu więcej niż jednego wątku.

  Zadanie ustalenia najbardziej optymalnych wartości parametrów `population_size`, `crossover_ratio`, `mutation_ratio`, `selection_size` nie jest zadaniem łatwym. Jednym z rozwiązań tego problemu jest wykorzystanie algorytmu genetycznego do ustalenia wartości parameterów, które gwarantują najskuteczniejsze działanie.

- Szybkość zmian najlepszego rozwiązania maleje wraz ze wzrostem liczby generacji. Wynika to z implementacji algorytmu a dalej - z ograniczeń zastosowanego kodowania.

- Działanie algorytmu cechuje duża losowość. Przy tych samych wartościach początkowych parametrów oczekiwane rezulaty mogą zostać uzyskane wcześniej lub później, tj. po mniejszej lub większej liczbie pokoleń.

- Pewne szczegóły implementacji operatora mutacji mają istotny wpływ na sprawność algorytmu; w szczególności zwrócenie uwagi na reprezentację problemu. W programie nie zostało wykorzystane najprostsze rozwiązanie, tj. losowa zmiana bitów w całym chromosomie; zamiast tego w jednym chromosomie zmieniane są bity należące tylko do części, która reprezentuje zmienną x lub tylko bity należące tylko do części, która reprezentuje zmienną y. Dzięki temu algorytm szybciej zbliża się do optymalnego rozwiązania.

- Pewne szczegóły implementacji operatora krzyżowania nie mają istotnego wpływu na sprawność algorytmu; w szczególności wybór liczby punktów przecięcia. Ad hoc zaimplementowana została funkcja, która dokonuje dwupuntkowego krzyżowania. Nie zaobserwowano jednak istotnej poprawy w

działaniu algorytmu. Stąd w projecie pozostawiono jedynie funkcję, która dokonuje jednopunktowe krzyżowania - z uwagi na zwiększenie prostoty programu.

- Wybór sposobu w jaki chromosomy są zakodowanie jest istotny dla sprawności algorytmu. Wykorzystano kodowanie binarnie (standard IEEE 754). Wtedy cecha nie wykracza poza zakres [-127, 128]. Jest ważne by zakres nie był zbyt szeroki, ponieważ szukane minima dla typowych funkcji optymalizujących są zazwyczaj rzędu jedności.

generacja	f(x,y)	x	y
0	-1.01924	0.000320186	-1.9395
1	-1.01924	0.000320186	-1.9395
2	-1.01929	0.000320186	-1.93947
3	-1.01929	0.000320186	-1.93947
4	-1.01929	0.000320186	-1.93947
5	-1.29354	0.000320186	-1.752
6	-1.29354	0.000320186	-1.752
7	-1.29354	0.000320186	-1.752
8	-1.29354	0.000320186	-1.752
9	-1.29354	0.000320186	-1.752
10	-1.447	3.52923e-15	-1.58959
11	-1.447	3.52923e-15	-1.58959
12	-1.447	3.52923e-15	-1.58959
13	-1.447	3.52923e-15	-1.58959
14	-1.447	3.52923e-15	-1.58959
15	-1.49847	3.52923e-15	-1.21458
16	-1.49847	3.52923e-15	-1.21458
17	-1.49847	3.52923e-15	-1.21458
18	-1.49847	3.52923e-15	-1.21458
19	-1.49847	3.52923e-15	-1.21458
20	-1.49847	3.52923e-15	-1.21458
21	-1.52813	-2.15806e-11	-1.34634
22	-1.52813	-2.15806e-11	-1.34634
23	-1.52813	-2.15806e-11	-1.34634
24	-1.52813	-2.15806e-11	-1.34634
25	-1.53628	-0.00579301	-1.34634
26	-1.53628	-0.00579301	-1.34634
27	-1.59051	-0.0463364	-1.34634
28	-1.59051	-0.0463364	-1.34634
29	-1.59051	-0.0463364	-1.34634
30	-1.59051	-0.0463364	-1.34634
31	-1.59051	-0.0463364	-1.34634
32	-1.59051	-0.0463364	-1.34634
33	-1.59051	-0.0463364	-1.34634
34	-1.59051	-0.0463364	-1.34634
35	-1.59051	-0.0463364	-1.34634
36	-1.59051	-0.0463364	-1.34634
37	-1.59051	-0.0463364	-1.34634
38	-1.6004	-0.0541487	-1.34634
39	-1.6004	-0.0541487	-1.34634
40	-1.6004	-0.0541487	-1.34634

Tablica 1: Statystyka dla generacji 0–40

generacja	f(x,y)	x	y
50	-1.78008	-0.710385	-1.34634
100	-1.90432	-0.632245	-1.56727
150	-1.90905	-0.540452	-1.49107
200	-1.90905	-0.540452	-1.49107
250	-1.90905	-0.540452	-1.49107
300	-1.90976	-0.4942	-1.52204
350	-1.90976	-0.4942	-1.52204
400	-1.9098	-0.49501	-1.53509
450	-1.9098	-0.49501	-1.53509
500	-1.9098	-0.49501	-1.53509
550	-1.9098	-0.49501	-1.53509
600	-1.9098	-0.49501	-1.53509
650	-1.9098	-0.49501	-1.53509
700	-1.9098	-0.49501	-1.53509
750	-1.9098	-0.49501	-1.53509
800	-1.9098	-0.49501	-1.53509
850	-1.91267	-0.566533	-1.56344
900	-1.91278	-0.562974	-1.56344
950	-1.91278	-0.562974	-1.56344
1000	-1.91278	-0.562974	-1.56344

Tablica 2: Statystyka dla generacji 50–1000

generacja	f(x,y)	x	y
1000	-1.91278	-0.562974	-1.56344
1500	-1.91319	-0.542228	-1.54724
2000	-1.91319	-0.542228	-1.54724
2500	-1.91322	-0.546061	-1.54525
3000	-1.91322	-0.545556	-1.54736
3500	-1.91322	-0.548357	-1.54719
4000	-1.91322	-0.548357	-1.54719
4500	-1.91322	-0.548357	-1.54719
5000	-1.91322	-0.548357	-1.54719
5500	-1.91322	-0.548357	-1.54719
6000	-1.91322	-0.547401	-1.54614
6500	-1.91322	-0.547401	-1.54614
7000	-1.91322	-0.547401	-1.54614
7500	-1.91322	-0.54676	-1.54598
8000	-1.91322	-0.547246	-1.54731
8500	-1.91322	-0.547246	-1.54731
9000	-1.91322	-0.547246	-1.54731
9500	-1.91322	-0.547246	-1.54731
10000	-1.91322	-0.547246	-1.54731

Tablica 3: Statystyka dla generacji 1000–10000