Jakub Latawiec

Rafał Malik

**Optymalizacja**

Laboratorium – Optymalizacja metodami niedeterministycznymi.

1. **Cel ćwiczenia**

Celem ćwiczenia jest zapoznanie się z niedeterministycznymi metodami optymalizacji poprzez ich implementację oraz wykorzystanie do wyznaczenia minimum podanej funkcji celu.

1. **Przeprowadzenie ćwiczenia**
2. Implementacja algorytmu ewolucyjnego

solution EA(matrix(\*ff)(matrix, matrix, matrix), int N, matrix lb, matrix ub, int mi, int lambda, matrix sigma0, double epsilon, int Nmax, matrix ud1, matrix ud2)

{

try

{

solution\* population = new solution[mi + lambda]; // Populacja (rodzice + potomkowie)

solution\* best\_mu = new solution[mi]; // Najlepsi osobnicy

matrix inv\_fitness(mi, 1), temp\_individual(N, 2);

double r, cumulative\_sum, total\_inv\_fitness;

double tau = 1.0 / sqrt(2 \* N), tau\_prime = 1.0 / sqrt(2 \* sqrt(N)); // Współczynniki mutacji

int worst\_idx; // Indeks najgorszego osobnika

// Inicjalizacja populacji początkowej

for (int i = 0; i < mi; ++i)

{

population[i].x = matrix(N, 2);

for (int j = 0; j < N; ++j)

{

population[i].x(j, 0) = (ub(j) - lb(j)) \* m2d(rand\_mat()) + lb(j);

population[i].x(j, 1) = sigma0(0);

}

population[i].fit\_fun(ff, ud1, ud2);

if (population[i].y < epsilon)

{

population[i].flag = 1;

return population[i];

}

}

// Główna pętla ewolucji

while (true)

{

total\_inv\_fitness = 0;

for (int i = 0; i < mi; ++i)

{

inv\_fitness(i) = 1 / population[i].y(0);

total\_inv\_fitness += inv\_fitness(i);

}

// Selekcja rodziców

for (int i = 0; i < lambda; ++i)

{

r = total\_inv\_fitness \* m2d(rand\_mat());

cumulative\_sum = 0;

for (int j = 0; j < mi; ++j)

{

cumulative\_sum += inv\_fitness(j);

if (r <= cumulative\_sum)

{

population[mi + i] = population[j];

break;

}

}

}

// Mutacja potomków

for (int i = 0; i < lambda; ++i)

{

r = m2d(randn\_mat());

for (int j = 0; j < N; ++j)

{

population[mi + i].x(j, 1) \*= exp(tau\_prime \* r + tau \* m2d(randn\_mat()));

population[mi + i].x(j, 0) += population[mi + i].x(j, 1) \* m2d(randn\_mat());

}

}

// Krzyżowanie

for (int i = 0; i < lambda; i += 2)

{

r = m2d(rand\_mat());

temp\_individual = population[mi + i].x;

population[mi + i].x = r \* population[mi + i].x + (1 - r) \* population[mi + i + 1].x;

population[mi + i + 1].x = r \* population[mi + i + 1].x + (1 - r) \* temp\_individual;

}

// Ocena funkcji celu dla potomków

for (int i = 0; i < lambda; ++i)

{

population[mi + i].fit\_fun(ff, ud1, ud2);

if (population[mi + i].y < epsilon)

{

population[mi + i].flag = 1;

return population[mi + i];

}

}

// Selekcja mi najlepszych osobników

for (int i = 0; i < mi; ++i)

{

worst\_idx = 0;

for (int j = 1; j < mi + lambda; ++j)

{

if (population[worst\_idx].y > population[j].y)

worst\_idx = j;

}

best\_mu[i] = population[worst\_idx];

population[worst\_idx].y = 1e10;

}

// Aktualizacja populacji bazowej

for (int i = 0; i < mi; ++i)

population[i] = best\_mu[i];

if (solution::f\_calls > Nmax)

break;

}

population[0].flag = 0;

return population[0];

}

catch (string ex\_info)

{

throw ("solution EA(...):\n" + ex\_info);

}

}

1. Implementacja testowej funkcji celu

matrix ff6T(matrix x, matrix ud1, matrix ud2)

{

return pow(x(0), 2) + pow(x(1), 2) - cos(2.5 \* M\_PI \* x(0)) - cos(2.5 \* M\_PI \* x(1)) + 2;

}

1. Implementacja funkcji różniczkowej dla problemu rzeczywistego

matrix df6(double t, matrix Y, matrix ud1, matrix ud2)

{

matrix dY(4, 1);

const double m\_1 = 5.0;

const double m\_2 = 5.0;

const double k\_1 = 1.0;

const double k\_2 = 1.0;

const double F = 1.0;

double b\_1 = ud2(0);

double b\_2 = ud2(1);

dY(0) = Y(1);

dY(1) = (-b\_1 \* Y(1) - b\_2 \* (Y(1) - Y(3)) - k\_1 \* Y(0) - k\_2 \* (Y(0) - Y(2))) / m\_1;

dY(2) = Y(3);

dY(3) = (F + b\_2 \* (Y(1) - Y(3)) + k\_2 \* (Y(0) - Y(2))) / m\_2;

return dY;

}

1. Implementacja funkcji celu dla problemu rzeczywistego

matrix ff6R(matrix x, matrix ud1, matrix ud2)

{

matrix y = 0;

matrix Y0 = matrix(4, new double[4] {0.0, 0.0, 0.0, 0.0});

matrix\* Y = solve\_ode(df6, 0.0, 0.1, 100.0, Y0, ud1, x[0]);

for (int i = 0; i < 1001; ++i)

{

y = y + abs(ud2(i, 0) - Y[1](i, 0)) + abs(ud2(i, 1) - Y[1](i, 2));

}

y(0) = y(0) / (2 \* ud1(0));

return y;

}

1. Implementacja funkcji lab6()

void lab6()

{

#ifdef SAVE\_TO\_FILE

create\_environment("lab06");

#endif

//Dane dokładnościowe

double epsilon = 1E-4;

int Nmax = 10000;

#ifdef CALC\_TEST

//Stringstream do zapisu danych

std::stringstream test\_ss;

//Rozwiązanie dla wyników testowych

solution test\_sol;

//Punkty startowe

matrix test\_x0{};

//Długości kroków

double sigma\_arr[] = { 0.01, 0.1, 1.0, 10.0, 100.0 };

//Zakresy

matrix lb\_test = matrix(2, new double[2] {-5.0, -5.0});

matrix ub\_test = matrix(2, new double[2] {5.0, 5.0});

//Liczba populacji

int mi\_test = 20;

int lambda\_test = 40;

for (auto sigma : sigma\_arr)

{

for (int i = 0; i < 100; ++i)

{

test\_sol = EA(ff6T, 2, lb\_test, ub\_test, mi\_test, lambda\_test, sigma, epsilon, Nmax);

test\_ss << test\_sol.x(0) << ";" << test\_sol.x(1) << ";" << test\_sol.y(0) << ";" << test\_sol.f\_calls << ";" << (solution::f\_calls > Nmax ? "NIE" : "TAK") << "\n";

solution::clear\_calls();

}

}

#ifdef SAVE\_TO\_FILE

save\_to\_file("test.csv", test\_ss.str());

#endif

#endif

#ifdef CALC\_SIMULATION

//Parametry zadania

matrix lb = matrix(2, new double[2] {0.1, 0.1});

matrix ub = matrix(2, new double[2] {3.0, 3.0});

matrix sigma0 = matrix(2, new double[2] {0.1, 0.1});

int mi = 50;

int lambda = 100;

//Pobieranie danych z pliku

matrix x1\_x2\_data = file\_reader::fileToMatrix(1001, 2, "../input\_data/lab06/polozenia.txt");

solution opt = EA(ff6R, 2, lb, ub, mi, lambda, sigma0, epsilon, Nmax, 1001, x1\_x2\_data);

std::cout << opt << "\n";

matrix y;

matrix Y0 = matrix(4, new double[4] {0.0, 0.0, 0.0, 0.0});

matrix\* Y = solve\_ode(df6, 0.0, 0.1, 100.0, Y0, NAN, opt.x[0]);

#ifdef SAVE\_TO\_FILE

save\_to\_file("simulation.csv", Y[1]);

#endif

#endif

}

1. **Parametry algorytmów**
2. Dla testów:

Dokładność (epsilon): 1E-4

Maksymalna ilość wywołań funkcji (Nmax): 10 000

Intensywność mutacji (sigma): [0.01, 0.1, 1.0, 10.0, 100.0]

Ograniczenia: (lb i ub): [-5.0, 5.0]

Liczba rodziców w populacji bazowej (mi): 20

Liczba potomków (lambda): 40

1. Dla problemu rzeczywistego:

Dokładność (epsilon): 1E-4

Maksymalna ilość wywołań funkcji (Nmax): 10 000

Intensywność mutacji (sigma): 0.1

Ograniczenia: (lb i ub): [0.1, 3.0]

Liczba rodziców w populacji bazowej (mi): 50

Liczba potomków (lambda): 100

1. **Dyskusja wyników**
2. Wyniki testowe:

Po przeprowadzeniu 100 optymalizacji dla różnych wartości intensywności mutacji i odrzuceniu wyników nie znajdujących minimów, otrzymaliśmy następujące wartości średnie:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Początkowa wartość zakresu mutacji | x1\* | x2\* | y\* | Liczba wywołań funkcji celu | Liczba minimów globalnych |
| 0,01 | -1,99E-05 | -1,66E-05 | 4,82E-05 | 400 | 68 |
| 0,1 | 1,87E-04 | -1,52E-04 | 4,95E-05 | 603 | 91 |
| 1 | 1,74E-05 | 1,30E-05 | 4,68E-05 | 984 | 99 |
| 10 | -1,55E-04 | -1,66E-05 | 5,19E-05 | 1884 | 97 |
| 100 | 1,18E-05 | 2,70E-05 | 4,86E-05 | 5544 | 84 |

Z powyższych danych widzimy że zwiększanie wartości epsilon, zwiększa ilość znajdowanych minimów globalnych kosztem zwiększenia ilości obliczeń. Wyjątkiem jest epsilon równy 100, gdzie skuteczność zaczyna maleć a ilość wywołań funkcji rośnie. Najbardziej optymalną wartością intensywności mutacji jest 10, w kontekście znalezienia najwięcej minimów. Dodatkowo średnia wartość funkcji jest najmniejsza dla epsilon = 10.

1. Wyniki problemu rzeczywistego:

Za pomocą algorytmu ewolucyjnego udało znaleźć się współczynniki oporu ruchu dla obu ciężarków. Wynoszą one: b1 = 1.5, b2 = 2.5. Po wykonaniu symulacji ruchu obu ciężarków dla znalezionych współczynników i porównaniu ich z wartościami z doświadczenia, otrzymaliśmy wykres:

Na wykresie nie są widoczne linie związane z danymi z doświadczenia, co oznacza że nasze dane z symulacji pokrywają się z danymi z doświadczenia z całkiem dużą dokładnością.

1. **Wnioski**

Skuteczność przeprowadzania optymalizacji niedeterministyczną metodą ewolucji zależy od dobrania odpowiedniego współczynnika intensywności mutacji. Na dokładność wyników może wpływać również ilość osobników w populacji ich potomków.