Московский государственный университет
имени М. В. Ломоносова
Факультет вычислительной математики и кибернетики

Задание 2. Генетические алгоритмы.

Отчёт
О выполненном задании

Выполнил:
студент 523 группы
Латыпов Ш. И.
latypovshamil2001@gmail.com

Москва
2023

Описание задачи

Решается задача минимизации непрерывной n-мерной функции f(x) с помощью генетического алгоритма. В качестве целевых функций выступают сферическая функция (f0), функция Розенброка (f1) и функция Растригина (f2):

$$f_0(x) = \sum_{i=1}^{n} x_i^2,$$

$$f_1(x) = \sum_{i=1}^{n-1} [100(x_i^2 - x_{i+1})^2 + (x_i - 1)^2],$$

$$f_2(x) = \sum_{i=1}^{n} [x_i^2 - 10cos(2\pi x_i) + 10].$$

Начальная область поиска $x_i \in [-100, 100], i = 1, ..., n$.

Имеется последовательная версия генетического алгоритма, в которой используются следующие генетические операторы:

- Турнирная схема отбора — все особи разбиваются на пары, в каждой паре с некоторой вероятностью производится турнир, в котором с заданной вероятностью побеждает особь с меньшим значением целевой функции. Побежденный удаляется из популяции, вместо него помещается копия победителя.
- Три варианта оператора скрещивания — одноточечное, двухточечное и равномерное.
- Мутация — с некоторой вероятностью к каждой компоненте решения (гену) прибавляется небольшое случайное число.

Для распараллеливания заданного генетического алгоритма предлагается использовать островную модель, когда вся популяция разбивается на субпопуляции (острова), внутри каждой субпопуляции реализуется описанный генетический алгоритм. С некоторой частотой между субпопуляциями выполняется циклическая миграция — две субпопуляции обмениваются частью своих особей, причём обемн происходит между субпопуляциями с номерами i и i + 1 (при этом первая субпопуляция взаимодействует с последней).

Параметры управляют работой миграции: размер миграции — сколько особей пересылается из одной субпопуляции в другую, это значение установлено в 5% от количества особей в одной субпопуляции, и частота миграции — количество итераций генетического алгоритма, после выполнения которых к субпопуляции применяется оператор миграции.

Задачей является MPI-реализация описанной островной модели. Параллельная программа должна корректно реализовывать генетический алгоритм и на выходе выдавать зависимость средней погрешности (т.е. разности между достигнутым минимальным значением целевой функции и значением ее глобального минимума) по популяции от номера итерации.

Код программы

```
#include <mpi.h>
#include <algorithm>
#include <cmath>
#include <iostream>
#include <vector>

using namespace std;
```

```
double frand() {
  static std::random_device rd;
  static std::mt19937 gen(rd());
  static std::uniform_real_distribution<> dis(0.0, 1.0);
  return dis(gen);
}

double gen(const double start, const double end) {
  static std::random_device rd;
  static std::mt19937 gen(rd());
  std::uniform_real_distribution<> dis(start, end);
  return dis(gen);
}

double sphere_function(double* x, int n) {
  double sum = 0;
  for (int i = 0; i < n; i++) {
    sum += x[i] * x[i];
  }
  return sum;
}

// Функция Розенброка
double rosenbrock_function(double* x, int n) {
  double sum = 0;
  for (int i = 0; i < n - 1; i++) {
    sum += 100 * (x[i + 1] - x[i] * x[i]) * (x[i + 1] - x[i] * x[i]) +
           (1 - x[i]) * (1 - x[i]);
  }
  return sum;
}

// Функция Растригина
double rastrigin_function(double* x, int n) {
  double sum = 10 * n;
  for (int i = 0; i < n; i++) {
    sum += x[i] * x[i] - 10 * (cos(2 * M_PI * x[i]));
  }
  return sum;
}

double eval(double* a, int n, int function_id = 1) {
  switch (function_id) {
    case 0:
      return sphere_function(a, n);
    case 1:
      return rosenbrock_function(a, n);
    case 2:
      return rastrigin_function(a, n);
    default:
      return sphere_function(a, n);
  }
```

```
  return 0;
}

void init(double* P, int m, int n) {
  for (int k = 0; k < m; k++) {
    for (int i = 0; i < n; i++) {
      P[k * n + i] = gen(-100, 100);
    }
  }
}

void shuffle(double* P, int m, int n) {
  for (int k = 0; k < m; k++) {
    int l = rand() % m;
    for (int i = 0; i < n; i++)
      swap(P[k * n + i], P[l * n + i]);
  }
}

void select(double* P, int m, int n) {
  double pwin = 0.9;
  shuffle(P, m, n);
  for (int k = 0; k < m / 2; k++) {
    int a = 2 * k;
    int b = 2 * k + 1;
    double fa = eval(P + a * n, n);
    double fb = eval(P + b * n, n);

    double p = frand();
    if ((fa < fb && p < pwin) || (fa > fb && p > pwin))
      for (int i = 0; i < n; i++)
        P[b * n + i] = P[a * n + i];
    else
      for (int i = 0; i < n; i++)
        P[a * n + i] = P[b * n + i];
  }
}

void crossover(double* P, int m, int n) {
  shuffle(P, m, n);
  for (int k = 0; k < m / 2; k++) {
    int a = 2 * k;
    int b = 2 * k + 1;
    int j = rand() % n;
    for (int i = j; i < n; i++)
      swap(P[a * n + i], P[b * n + i]);
  }
}

void mutate(double* P, int m, int n) {
  double pmut = 0.05;
  double mutation_strength = 5;
```

```
  for (int k = 0; k < m; k++) {
    for (int i = 0; i < n; i++) {
      if (frand() < pmut) {
        P[k * n + i] += (frand() * 2.0 * mutation_strength) - mutation_strength;
        // Ограничить значения в диапазоне [-100, 100]
        P[k * n + i] = std::max(-100.0, std::min(100.0, P[k * n + i]));
      }
    }
  }
}

double printthebest(double* P, int m, int n) {
  int k0 = -1;
  double f0 = 1e12;
  for (int k = 0; k < m; k++) {
    double f = eval(P + k * n, n);
    if (f < f0) {
      f0 = f;
      k0 = k;
    }
  }
  // cout << f0 << ": ";
  // for (int i = 0; i < n; i++)
  //   cout << P[k0 * n + i];
  // cout << endl;
  return f0;
}

void migrate(double* P, int m, int n, int world_rank, int world_size) {
  MPI_Status status;

  int next = (world_rank + 1) % world_size;
  int prev = (world_rank - 1 + world_size) % world_size;

  // Определение размера миграции
  int migration_size = m / 20;

  double* individuals_to_send = new double[migration_size * n];
  double* individuals_to_recv = new double[migration_size * n];

  for (int i = 0; i < migration_size * n; ++i) {
    individuals_to_send[i] = P[i];
  }

  MPI_Sendrecv(individuals_to_send, migration_size * n, MPI_DOUBLE, next, 0,
               individuals_to_recv, migration_size * n, MPI_DOUBLE, prev, 0,
               MPI_COMM_WORLD, &status);
  MPI_Barrier(MPI_COMM_WORLD);

  for (int i = 0; i < migration_size * n; ++i) {
    P[i] = individuals_to_recv[i];
  }
```

```
  delete[] individuals_to_send;
  delete[] individuals_to_recv;
}

void runGA(int n, int m, int T, int world_rank, int world_size) {
  int local_m = m / world_size;
  double* P = new double[n * local_m];
  double* global_P = nullptr;

  if (world_rank == 0) {
    global_P = new double[n * m];
    init(global_P, m, n);
  }

  MPI_Scatter(global_P, n * local_m, MPI_DOUBLE, P, n * local_m, MPI_DOUBLE, 0,
              MPI_COMM_WORLD);

  if (world_rank == 0) {
    delete[] global_P;
  }

  MPI_Barrier(MPI_COMM_WORLD);

  double best = 0.;
  double sum = 0.;
  double totalsum = 0.;
  double totalbest = 0.;

  for (int t = 0; t < T; t++) {
    select(P, local_m, n);
    crossover(P, local_m, n);
    mutate(P, local_m, n);

    best = printthebest(P, local_m, n);
    sum = best;
    MPI_Allreduce(&sum, &totalsum, 1, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);
    MPI_Allreduce(&best, &totalbest, 1, MPI_DOUBLE, MPI_MIN, MPI_COMM_WORLD);
    if (world_rank == 0) {
      cout << t << " " << totalbest << " " << totalsum / world_size << endl;
    }

    if (t % 20 == 0) {  // Миграция каждые 20 итераций
      migrate(P, local_m, n, world_rank, world_size);
      MPI_Barrier(MPI_COMM_WORLD);
    }
  }

  delete[] P;
}

int main(int argc, char** argv) {
```

```
  MPI_Init(&argc, &argv);

  int world_size;
  MPI_Comm_size(MPI_COMM_WORLD, &world_size);

  int world_rank;
  MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);

  int n = 500;
  int m = 400;
  int T = 5000;

  if (m % world_size != 0) {
    if (world_rank == 0) {
      std::cerr << "Ошибка: общий размер популяции должен быть кратен "
                   "количеству процессов MPI."
                << std::endl;
    }
    MPI_Finalize();
    return 1;
  }

  runGA(n, m, T, world_rank, world_size);

  MPI_Finalize();
  return 0;
}
```

Результаты работы программы

Запуск программы происходил с параметрами:
n = 500
m = 400
T = 1000
Частота миграции = 20
Количество Миграции = 5% от числа особей в субпопуляции
Количество MPI процессов = 4
Графики имеют логарифмическую шкалу по Оси y.

Сферическая функция

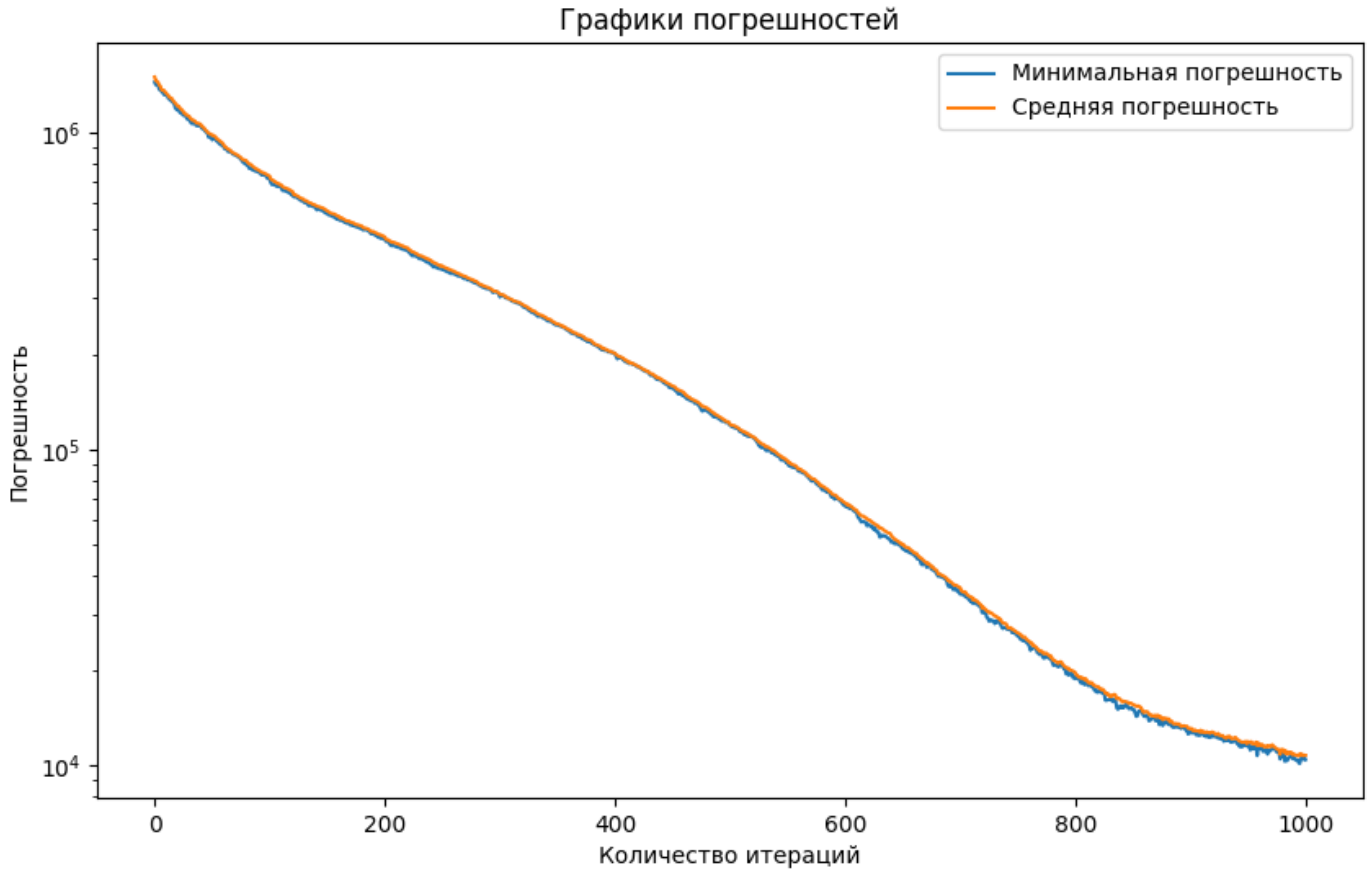

Рис. 1: Сферическая функция

Функция Розенброка

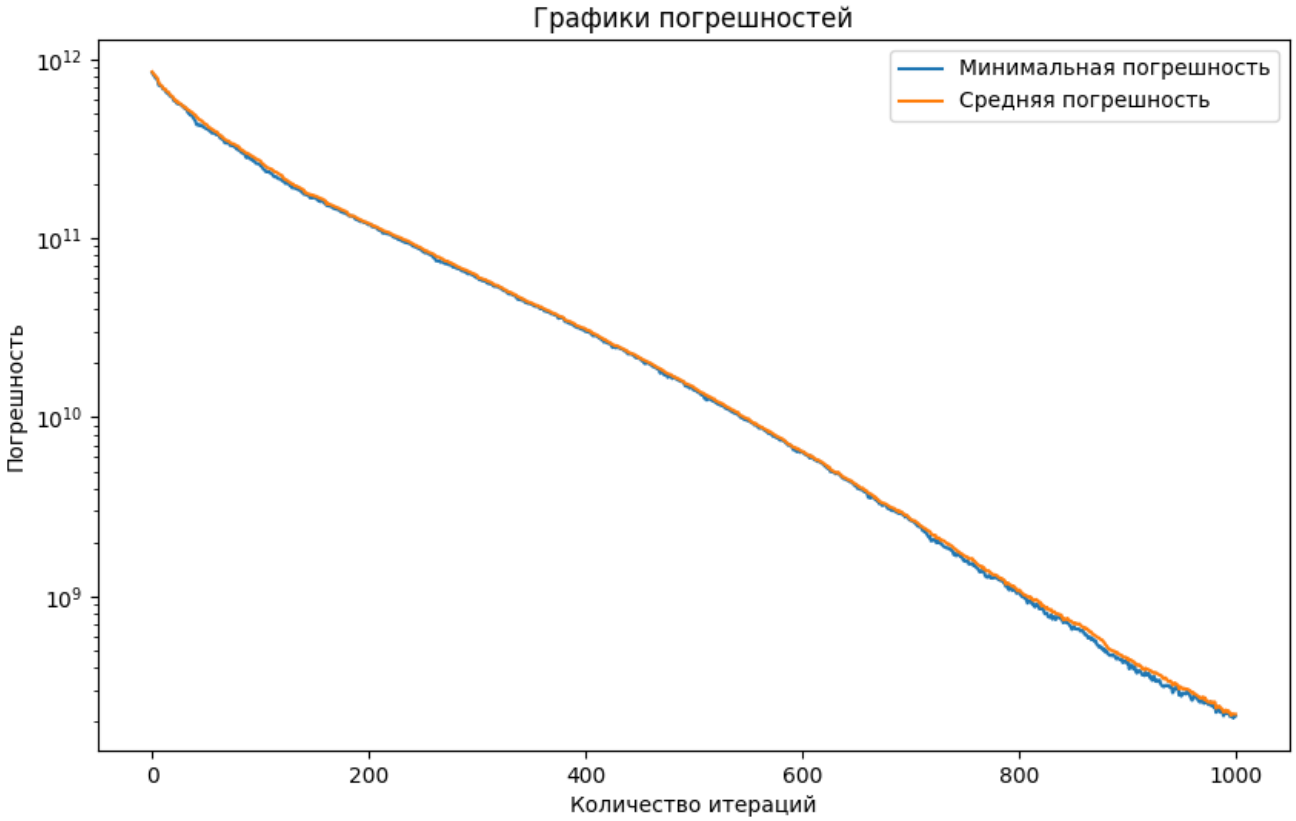

Рис. 2: Функция Розенброка

Функция Растригина

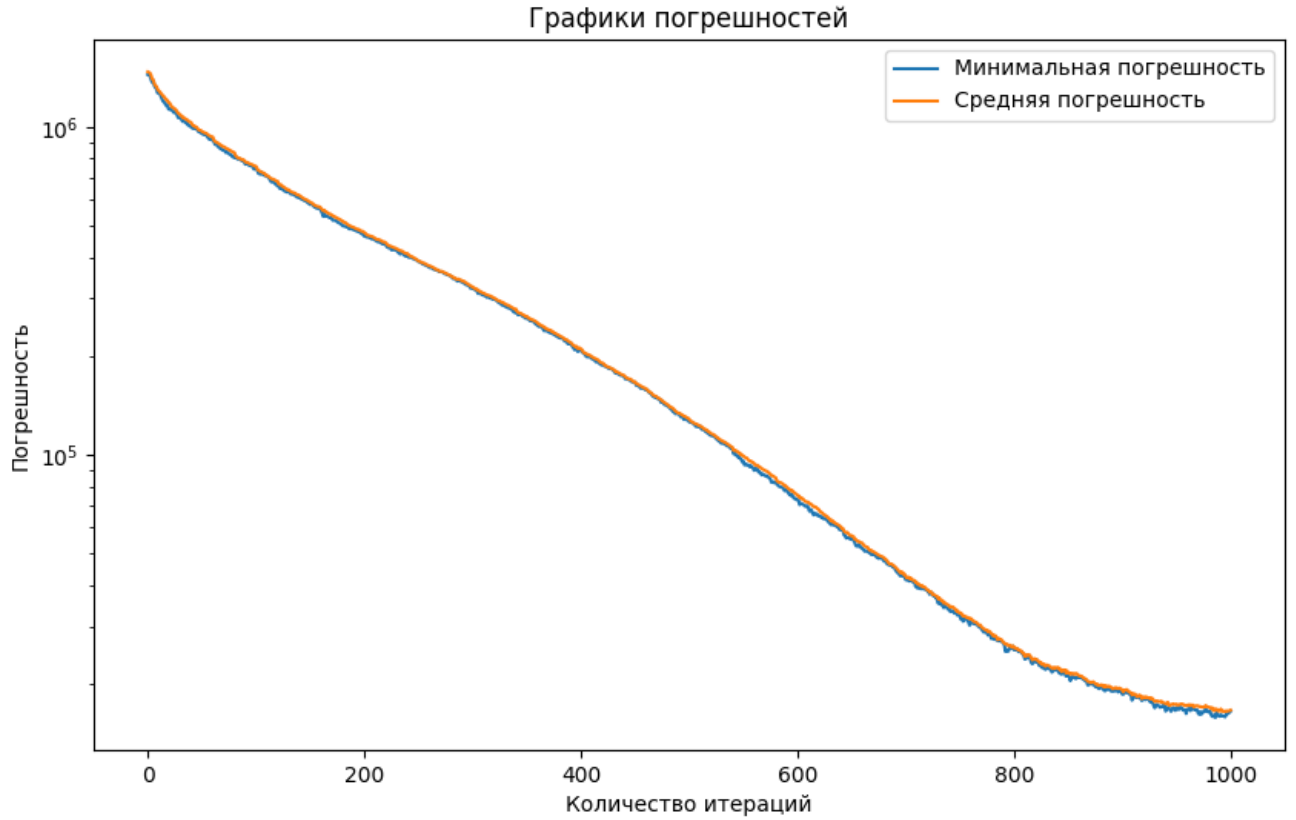

Рис. 3: Функция Растригина