**Мавроматидис В. Ю.**

**Научный руководитель: ассистент каф. САПР Данильченко Е. В.**

Южный федеральный университет

ИКТИБ, Таганрог

[mavromatidis@sfedu.ru](mailto:mavromatidis@sfedu.ru)

**РАЗРАБОТКА ГЕНЕТИЧЕСКОГО АЛГОРИТМА ДЛЯ МОДУЛЯ ПРОГОЗИРОВАНИЯ НЕИСПРАВНОСТЕЙ ВЕТРОГЕНЕРАТОРОВ**

*В статье рассматривается применение алгоритмов на основе эволюционных методов в сфере прогнозирования неисправностей на ветрогенераторных электростанциях. Производится анализ эффективности генетического алгоритма в прогнозировании по сравнению с известными методами.*

*ВЕТРОГЕНЕРАТОРЫ, БИОИНСПИРИРОВАННЫЕ АЛГОРИТМЫ, ПРОГНОЗИРОВАНИЕ, ЭВОЛЮЦИОННЫЕ МЕТОДЫ, ГЕНЕТИЧЕСКИЙ АЛГОРИТМ.*

В настоящее время для обслуживания ветрогенераторов привлекаются альпинисты для визуальных осмотров и используются беспилотные летательные аппараты, управляемые с земли. Хотя эти методы выполняют свою функцию, они требуют значительных временных и финансовых затрат, а также не обеспечивают возможности прогнозирования неисправностей и их своевременного устранения.

В рамках проекта “Специализированные дроны для прогнозного обслуживания ветрогенераторов” для решения данной проблемы были проведены исследования возможностей применения биоинспирированных алгоритмов на основе эволюционных вычислений для прогнозирования возникновения эксплуатационный неисправностей на ветрогенераторных электростанциях.

Генетический алгоритм (ГА) — это эвристический алгоритм поиска, основанный на применении методов естественного отбора и эволюции, и используемый для решения задач моделирования, оптимизации и прогнозирования.

Ключевые функции генетического алгоритма:

1. Создание первоначальной популяции;
2. Мутация;
3. Кроссигновер;
4. Вычисление приспособленности (fitness-функция);
5. Турнирный отбор.

Для более подробного описания способа реализации данных методов в разработанном алгоритме приведен псевдокод:

|  |
| --- |
| Псевдокод 1. Создание начальной популяции |
| vector<vector<NodeState>> initialize\_population(int size, int num\_nodes) {  vector<vector<NodeState>> population(size, vector<NodeState>(num\_nodes)); //Создаем вектор для хранения популяции  for (int i = 0; i < size; ++i) {  for (int j = 1; j < num\_nodes; ++j) {  population[i][j] = {  rand() % 5, // blades\_mechanical\_damage  rand() % 5, // flaps\_condition  rand() % 5, // corrosion  rand() % 5, // icing  rand() % 5, // rotary\_mechanism  rand() % 5, // wind\_sensor  };  }//Заполняем параметры случайными числами от 0 до 5  }  return population;  } |

Первоначальная популяция формируется путем присвоения каждому показателю случайно выбранного целочисленного значения от 0 до 5.

|  |
| --- |
| Псевдокод 2. Мутация |
| void mutate(std::vector<NodeState>& individual, double mutation\_rate) {  for (size\_t i = 1; i < individual.size(); ++i) {  if (static\_cast<double>(rng()) / rng.max() < mutation\_rate) {  individual[i].flaps\_condition = 5 - individual[i].flaps\_condition;  individual[i].blades\_mechanical\_damage = 5 - individual[i].blades\_mechanical\_damage;  individual[i].corrosion = 5 - individual[i].corrosion;  individual[i].icing = 5 - individual[i].icing;  individual[i].rotary\_mechanism = 5 - individual[i].rotary\_mechanism;  individual[i].wind\_sensor = 5 - individual[i].wind\_sensor;  }  }  } |

Функция мутации принимает в качестве параметров одного из “индивидов”, и значение mutation\_rate, от которого зависит количество мутирующих параметров данного индивида. Далее случайно выбранные показатели меняются.

|  |
| --- |
| Псевдокод 3. Кроссинговер |
| std::vector<NodeState> crossover(const std::vector<NodeState>& parent1, const std::vector<NodeState>& parent2) {  std::vector<NodeState> child(parent1.size());  for (size\_t i = 1; i < parent1.size(); ++i) {  child[i] = (rng() % 2) ? parent1[i] : parent2[i];  }  return child;  } |

С помощью функции кроссинговера, представленной в псевдокоде 3, из двух родительских индивидов формируется индивид потомка. Случайно выбранные параметры в равном количестве наследуются от каждого из родителей.

|  |
| --- |
| Псевдокод 4. Fitness-функция |
| double fitness(const std::vector<NodeState>& individual, const std::vector<std::vector<int>>& G) {  double fitness\_value = 0.0;  for (int node = 1; node < G.size(); ++node) {  const auto& state = individual[node];  fitness\_value += state.flaps\_condition \* 3;  fitness\_value += state.blades\_mechanical\_damage \* 2;  fitness\_value += state.corrosion \* 1;  fitness\_value += state.icing \* 0.1;  fitness\_value += state.rotary\_mechanism \* 0.5;  fitness\_value += state.wind\_sensor \* 0.5;  }  return fitness\_value;  } |

В псевдокоде 4 показана целевая функция, которая определяет уровень “приспособленности индивида”. В данном случае функция fitness определяет, насколько оптимальным является каждое найденное решение.

vector<NodeState> tournament\_selection(const vector<vector<NodeState>>& population, const vector<double>& fitnesses, int k) {

Псевдокод 5. Турнирный отбор

vector<int> selected;

for (int i = 0; i < k; ++i) {

int ind = rng() % population.size();

selected.push\_back(ind);

}

int best = selected[0];

for (int i = 1; i < k; ++i) {

if (fitnesses[selected[i]] > fitnesses[best]) {

best = selected[i];

}

}

return population[best];

}

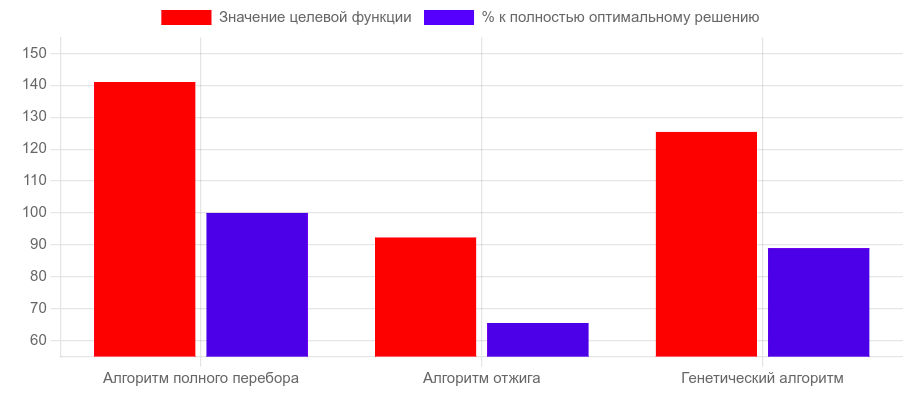
После выполнения всех предыдущих операций и вычисления приспособленности для индивида последующая популяция формируется методом турнирного отбора наиболее приспособленных индивидов, т. е. решений, показавших наиболее высокие значения целевой функции. Остальные, менее приспособленные индивиды, отсеиваются таким образом, чтобы размер популяции всегда оставался постоянным.

В ходе данного исследования был проведен эксперимент, в котором разработанный алгоритм сравнивался с известными методами решения подобных задач, такими как алгоритм полного перебора и алгоритм отжига.

Для эксперимента данные алгоритмы были применены для решения задачи нахождения оптимального значения целевой функции. В таблице 1, представленной ниже, собраны результаты работы каждого алгоритма.

Таблица 1 - Результаты исследования

|  |  |  |  |
| --- | --- | --- | --- |
| Название алгоритма | Значение целевой функции | % к полностью оптимальному решению | Время выполнения алгоритма |
| Алгоритм полного перебора | 141 | 100% | 0.07889 |
| Алгоритм отжига | 92.3 | 65.5% | 0.00131 |
| Генетический алгоритм | 125.4 | 88.9% | 0.00964 |

Рисунок 1 – Сравнение показателей целевой функции и процентое отношение к полностью оптимальному решению

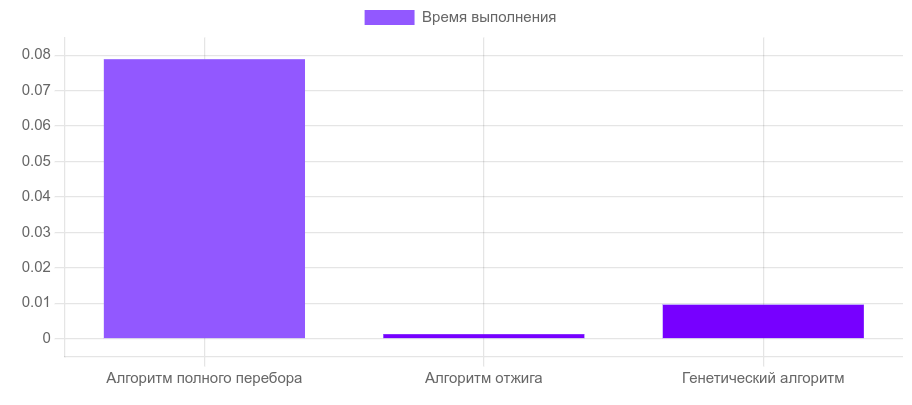


Рисунок 2 – Сравнение времени выполнения алгоритмов

**Заключение.** Результаты исследования показали, что разработанный генетический алгоритм способен находить значение, достаточно близкое к оптимальному, при затратах времени примерно в 8 раз меньших, чем потребовалось для поиска абсолютного оптимума методом полного перебора. Алгоритм отжига выполняется быстрее, чем алгоритм, основанный на эволюционных методах, но его точность гораздо ниже.

Таким образом, разработанный алгоритм может успешно применяться для решения задач, связанных с прогнозированием неисправностей ветрогенераторов, выдавая результат, близкий к оптимальному, за приемлемое время.

**Поддержка.** Исследование выполнено за счет гранта Российского научного фонда No 24-71-00035, https://rscf.ru/project/24-71-00035/ в Южном федеральном университете.

**Библиографический список**

1. А.С. Татевосян. Практикум по технологиям разработки алгоритмов интеллектуальных инфориационных систем: Методические указания для проведения лабораторных занятий по дисциплине «Интеллектуальные информационные системы» Омск: Изд-во ОмГТУ , 2008. – 60 с.