МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РФ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ  
ВЫСШЕГО ПРОФЕССИОНАЛЬНОГО ОБРАЗОВАНИЯ  
ВЯТСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

ФАКУЛЬТЕТ ПРИКЛАДНОЙ МАТЕМАТИКИ И  
ТЕЛЕКОММУНИКАЦИЙ

КАФЕДРА ПРИКЛАДНОЙ МАТЕМАТИКИ И ИНФОРМАТИКИ

**Кумулятивный многонишевый генетический алгоритм   
для оптимизации мультимодальных функций**

Пояснительная записка

Курсовая работа по дисциплине

«Модели и методы поисковой оптимизации»

ТПЖА.010541.012 ПЗ

Разработал студент гр. ПМ-41 \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ / Буров С. Д. /

(подпись)

Руководитель преподаватель \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ / Прозорова Т. Г. /

(подпись)

Проект защищен с оценкой «\_\_\_\_\_\_\_\_\_\_\_\_\_» «\_\_»\_\_\_\_\_\_\_ 2014 г.

Члены комиссии \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ /\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_/

(подпись)

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ /\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_/

(подпись)

Киров 2014

Содержание

[Введение 4](#_Toc378670056)

[1 Описание алгоритма 5](#_Toc378670057)

[1.1 Отбор и кроссовер 7](#_Toc378670058)

[1.2 Мутация 9](#_Toc378670059)

[1.3 Добавление 10](#_Toc378670060)

[1.4 Масштабирование приспособленности 11](#_Toc378670061)

[2 Сравнение с аналогичными методами 14](#_Toc378670062)

[2.1 Кумулятивный генетический алгоритм Зиона и Шнайдера 14](#_Toc378670063)

[2.2 Генетический алгоритм Гантовника 14](#_Toc378670064)

[2.3 Метод разделения 14](#_Toc378670065)

[2.4 Метод вытеснения 15](#_Toc378670066)

[2.5 Метод многонишевого вытеснения Седено 15](#_Toc378670067)

[2.6 Метод ограниченого конкурентного отбора 15](#_Toc378670068)

[2.7 Метод Ху и др. 16](#_Toc378670069)

[2.8 Метод Лиана и Леуна 16](#_Toc378670070)

[2.9 Метод Куэваса и Гонцалеса 16](#_Toc378670071)

[2.10 Итоги сравнения 17](#_Toc378670072)

[3 Особенности реализации алгоритма 18](#_Toc378670073)

[3.1 Хранение генетического кода 18](#_Toc378670074)

[3.2 Генерирование нормально распределённой случайной величины 20](#_Toc378670075)

[3.3 Выбор медианы массива и числа определённого ранга 20](#_Toc378670076)

[3.4 Пропорционалный приспособленности отбор 21](#_Toc378670077)

[4 Результаты тестирования 23](#_Toc378670078)

[Заключение 26](#_Toc378670079)

[Приложение А. Часть листинга программы 27](#_Toc378670080)

[Приложение Б. Перевод статьи 41](#_Toc378670081)

[Приложение В. Оригинал статьи 49](#_Toc378670082)

[Приложение Д. Библиографический список 63](#_Toc378670083)

## Введение

Оптимизация функций – общирная сфера деятельности, в которой принято выделять более узкие направления, среди которых есть и оптимизация мультимодальных функций. Выделение её в отдельный класс задач обусловлено ограничениями существующих методов локальной оптимизации. Среди методов решения задачи глобальной оптимизации существуют методы с общим названием – мета эвристики. Отличительной особенностью данных методов является то, что они способны находить локальные оптимумы и выходить из них, переходя к следующему. Таким образом алгоритмы данного типа способны самостоятельно находить гловальный оптимум среди множества локальных. К данному типу алгоритмов относятся генетические алгоритмы, мимикрирующие природные механизмы передачи наследственной информации. Будучи основаными на простейших базовых принципах отбора, кроссовера и мутаций данные алгоритмы способны тонко настраиваться под конкретную задачу и превосходить в её решении другие методы.

Кумулятивный многонишевый генетический алгоритм (A Cumulative Multi-Niching Genetic Algorithm) был разработан [1] для ускорения решения задач оптимизации мультимодальных целевых функций, имеющих высокую вычислительную сложность. Сохраняя всех особей в популяции, КМН ГА использует каждую оценку целевой функции для полчения информации о структуре пространства, а анализ распределения плотности популяции позволяет позволяет избежать излишних оценок целевой функции. Разработанный механизм генетических операций обеспечивает быструю и устойчивую сходимость к множеству локальных оптимумов, предоставляя возможность выбора необходимого локального оптимума по альтернативным, сложно формализуемым критериям.

## Описание алгоритма

Отличительной особенностью КМН ГА является его кумулятивность. Каждое последующее поколение добавляется в популяцию. С целью минимизации оценок фунции, особи, для которых целевая функция уже рассчитана, никогда не исключаются; даже неприспособленные особи имеют значение, сообщая алгоритму куда идти не нужно. Ключевым для работы кумулятивного метода является использование адаптивного ограничения близости, что предотвращает появление отпрысков, очень схожих с особями, существующими в популяции. Используя пороговое расстояние, обратно пропорциональное приспособленности соседних особей, КМН ГА поощряет сходимость вокруг перспективных регионов структуры пространства и поддерживает разреженную плотность популяции в низкоприспособленных регионах структуры пространства.

Фундаментальное отличие КМН ГА от других генетических алгоритмов в том, что он использует множество уникальных особенностей в генетических операциях, соединённых вместе (как приведено на Рисунке 2). Отбор и кроссовер разработаны для поддержания стабильных подпопуляций около локальных оптимумов и обеспечения сходимости алгоритма. Операция мутации разработана для поощрения многообразия и исследования структуры пространства. Операция «добавление», которая занимает место операции замены классического генетического алгоритма, разработана для обеспечения использования накопившейся популяции особей, для того чтобы избежать излишних или ненужных оценок целевой функции и руководить генетическим алгоритмом для создания отпрысков в наиболее перспективных областях структуры пространства. Операция масштабирования приспособленности позволяет генетическому алгоритму обрабатывать локальные оптимумы равнозначно несмотря на возможные различия в приспособленности.

В общем виде алгоритм имеет следующую схему (Рисунок 1):



Рисунок 1 - Общая схема КМН ГА

### Отбор и кроссовер

Процесс отбора и кроссовера объединяет пропорциональный приспособленности отбор и схему вытеснения-привлечения (crowding-inspired), смещённую в сторону соседних особей.

Первый родитель, P1, каждой пары выбирается из популяции, используя стандартный пропорциональный приспособленности отбор (FPS) с выроятностью выбора пропорциональной приспособленности. Затем, для каждого P1, из Ncrowd случайных кандидатов, пара выбирается, используя отбор, пропорциональный близости (PPS), где вероятность выбора определяется вероятностной функцией, описывающей, насколько близко кандидат P2 к P1 в структуре пространства. Наиболее простая вероятностная функция – инверсия расстояния по Евклиду:

(1)

где X – переменная-вектор длины n решения особи. Наиболее приспособленный из кандидатов выбирается в пару к P1. Этот процесс повторяется для каждой особи, выбранной в качестве P1–родителя для кроссовера.

Благодаря тому, что парой к особи является наиболее приспособленная из случайной группы кандидатов, в основном, соседних особей, поощряется кроссовер между особями из одной ниши, хотя вероятностный отбор группы позволяет редкие кроссоверы с удалёнными особями, обеспечивая важную возможность кроссовера между нишами. Этот подход способствует стабильной многонишевости КМН ГА и является основой кроссовер-сходимости популяции к локальным оптимумам.

В операции кроссовера генетический код отпрыска выбирается равномерно-случайно из гиперкуба, ограниченного генетическими кодами обоих родителей.

Схема кроссовера представлена ниже (Рисунок 2):



Рисунок 2 - Общая схема отбора и кроссовера

### Мутация

Операция мутации происходит паралельно с операцией кроссовера. Мутационный отбор выполняется случайно и мутация генетического кода каждой особи базируется на нормальном распределении. Это даёт алгоритму способность широкого изучения структуры пространства. Хотя приспособленность особей явно не используется в операции мутации, последующая операция добавления делает более вероятным, что мутация произойдёт в более приспособленных регионах структуры пространства.

Общая схема мутации представлена на рисунке 2:



Рисунок 3 - Общая схема мутации

### Добавление

Кумулятивная природа CMN GA исключает использование операции замены. Вместо этого, отпрыски добавляются к постоянно расширяющейся популяции. Ограничение близости гарантирует, что алгоритм сходится к более приспособленным особям и уходит от менее приспособленных. Эта фильтрация, имеющая место перед оценкой приспособленности отпрысков, ключевая для кумулятивного подхода к популяции. Исключая отпрысков, которые чрезмерно схожи с существующими членами популяции, избегаются излишние оценки целевой функции.

Пороговое расстояние ограничения близости, Rmin, обратно-пропорционально зависит от приспособленности соседних существующих особей, Fnearest, как определено функцией пороговой дистанции. Простым примером этого будет:

(2)

Значения этой функции пороговой дистанции будет 0.001 около наиболее приспособленных особей и 0.101 около наименее приспособленных особей, где дистанция нормализована границами структуры пространства, и приспособленность масштабирована к промежутку [0..1].

Такой подход для функции добавления позволяет потомкам быть очень близко к приспособленным особям и обеспечивает большую минимальную дистанцию для менее приспособленных особей. По существу, плотность популяции будет высокой в хороших и низкой в плохих регионах структуры пространства, как определено накопленными оценками целевой функции в течение работы генетического алгоритма. Карта плотности популяции, по существу, прописывает структуру пространства как прогресс алгоритма. Если структура пространства известна априори, использование сеточного исследования структуры пространства может быть более эффективным, но без этого знания более адаптивный метод будет более практичным.

Для приспособления алгоритма к изменяющимся задачам в ходе оптимизации – вначале для исследования структуры пространства, а затем для сужения на локальные оптимумы – в функции пороговой дистанции могут быть сделаны изменения в зависимости от количества особей или номера поколения, G. Это может помочь предотвратить преждевременную сходимость, обеспечив нахождение всех локальных оптимумов. Функция пороговой дистанции, которую использовал автор статьи [1], для получения результатов:

(3)

### Масштабирование приспособленности

Алгоритм, описанный до сих пор, потенциально может сойтись только к наиболее приспособленному локальному оптимуму. Последний компонент, разработанный для решения этой задачи и обеспечения одинакового отношения ко всем значимым локальным оптимумам, является операция близостно-весового масштабирования приспособленноти. В большинстве генетических алгоритмов функция масштабирования применяется к значениям приспособленности популяции для масштабирования их в нормализованные границы, а также чтобы регулировать распределение приспособленности. Простейший подход – линейно масштабировать значение приспособленности, F, в промежуток [0..1], так что наименьшее значение приспособленности спроецируется в F’=0 и наибольшее значение в F’=1:

(4)

Функция масштабирования также может быть использована для регулирования распределения приспособленности в ширину промежутка значений приспособленности для того, чтобы, например, обеспечить внимание более или менее к умерено приспособленным особям. Это масштабироване может быть адаптировано к особенностям популяции. Для получения результатов, представленных здесь, использовалась вторая экспоненциальная масштабирующая функция для регулирования значений масштабированной приспособленности так, чтобы медиана равнялась 0.5:

(5)

Близостно-весовое масштабирование приспособленности, ключевой компонент КМН ГА, добавляет дополнительную операцию масштабирования. Эта операция зависит от обнаружения локально-оптимальных особей в популяции. Использовался критерий, что особь считается представляющей локальный оптимум, если она приспособленнее чем все ближайшие Nmin соседи. В операции близостно-весового масштабирования масштабирующие функции (4) и (5) применяются к популяции множество раз, каждый раз нормализуя результаты к приспособленности различных локальных оптимумов. Так, если было найдено m локальных оптимумов, каждая особь популяции будет иметь m масштабированных значений приспособленности. Эти масштабированные значения приспособленности F’’ затем комбинируются для каждой особи i относительно близости особи к каждому соответствующему локальному оптимуму j для получения значений финальной масштабированой приспособленности популяции:

(6)

Близость, Pi,j, может быть расчитана, как в (1). Такой процесс даёт каждому локальному оптимуму равные значения приспособленности.

Общую схему можно увидеть на рисунке 4:



Рисунок 4 - Общая схема масштабирования приспособленности

## Сравнение с аналогичными методами

Для сравнения были выбраны другие многонишевые генетические алгоритмы. Особенности некоторых из них перешли в КМН ГА.

### Кумулятивный генетический алгоритм Зиона и Шнайдера

Кумулятивные алгоритм Зиона и Шнайдера (Xiong and Schneider) хранит всех особей с высоким значением приспособленности, для использования вместе с текущим поколением в размножении. Это достижение полезно для сохранения информации о лучшем регионе структуры пространства, но это не позволяет избежать лишних оценок целевой функции.

### Генетический алгоритм Гантовника

Генетический алгоритм, разработанный Гантовником (Gantovnik), в отличие от предыдущего, позволяет. Его алгоритм хранит информацию обо всех предыдущих особях и использует её для построения метода Шепарда (Shepard) реакции поверхностной аппроксимации окружающих значений приспособленности (Shepard’s method response surface approximation of surrounding fitness values), который может быть использован вместо оценки целевой функции для близких особей.

### Метод разделения

Метод разделения (Sharing), предложенный Холландом и раскрытый Голдбергом и Ричардсоном, снижает приспособленность каждой особи, относительно количества соседних особей. Снижение приспособленности определено функцией разделения, которая включает предельную дистанцию, определяющую какой уровень сходства составляют соседние особи. Недостатком этого метода является то, что для выбора хорошей функции разделения необходимо априори знать характеристики целевой функции. Также этот метод имеет трудности при формировании стабильной подпопуляции, хотя в этой области были сделаны улучшения.

### Метод вытеснения

Альтернативой является метод вытеснения (Crowding) Де Ёнга (De Jong), особенностью которого является замена шага, определяющего какие особи составят следующее поколение: для каждого потомка выбирается случайное подмножество особей из текущей популяции и особь из него, наиболее схожая с потомком, заменяется им. Улучшение, внесённое Махфоундом (Mahfoud) и названное детерминированное вытеснение (Deterministic crowding), убирающее давление отбора в репродукции использованием случайного, а не пропорционального приспособленности отбора и изменившее шаг замены так, что каждый кроссоверный потомок конкурирует с более похожим на него родителем за вхождение в следующее поколение.

### Метод многонишевого вытеснения Седено

Метод многонишевого вытеснения (Multi-Niche Crowding) метод Седено (Cedeño) отличается от предыдущих методов вытеснения реализацией принципа выталкивания на стадии отбора. Для каждой кроссовер-пары один родитель, выбирается случайно или последовательно, а другой как наиболее схожая с ним особь из группы случайно выбранных особей.

Это способствует кроссоверу между близлежащими особями, обеспечивая стабильность многонишевости. Операция замены описана как «худший среди наиболее схожих»; создаётся случайное количество групп из популяции, особи из каждой группы, наиболее схожие с рассматриваемым потомком выбираются и наименее подходящий из этих «наиболее схожих» особей заменяется потомком.

### Метод ограниченого конкурентного отбора

Хотя метод многонишевого вытеснения вполне эффективен в поиске множества локальных оптимумов, он и другие методы, описанные выше, всё ещё отдают предпочтение оптимуму с наибольшим значением приспособленности. Ли, Чо и Юн (Lee, Cho, and Jung) приводят другой метод, названный Ограниченый конкурентный отбор (Restricted Competition Selection), который превосходит предыдущие методы в нахождении и удержании даже слабого локального оптимума. В ином-классическом методе каждая пара особей внутри «нишевого радиуса» (niche radius) друг от друга сравниваются и приспособленность менее приспособленной особи устанавливается равной нулю. Это в сущности оставляет только локально-оптимальных особей для репродукции. Самые приспособленные из тех особей сохраняются в следующем поколении, как элита.

### Метод Ху и др.

Некоторые более поздние генетические алгоритмы используют информацию о направлении для для обеспечения лучшего управления в исследованием структуры пространства. Ху и др. (Hu et al.) зашли так далеко, что численно вычисляли градиент целевой функции для каждой особи для того, чтобы использовать метод наискорейшего спуска (steepest descent) для выбора потомка.

Этот метод мощный, но большое количество оценок функции делает его неприменимым для вычислительно-ёмких целевых функций.

### Метод Лиана и Леуна

Лиан и Леун (Liang and Leung) используют более умереный метод в котором два потенциальных потомка создаются на линии, соединяющей две существующих особи и четыре результирующие значения приспособленности сравниваются для того, чтобы прогнозировать положение соседних пиков. Используя эту информацию в особенно устроенный кроссовер и мутацию, алгоритм производит значительно меньше оценок функции, чем другие участвующие в сравнении генетические алгоритмы.

### Метод Куэваса и Гонцалеса

Метод, использующий ещё меньше оценок функции, - эволюционный алгоритм (evolutionary algorithm) Куэваса и Гонцалеса (Cuevas and Gonźalez), имитирующий коллелктивное поведение животных. Этот алгоритм моделирует путь животных влекомых или отталкиваемых от доминантных особей и сохраняет в памяти набор наиболее приспособленных особей. Конкуренция между особями, которые находятся ближе определённого порога, также включена. Не смотря на отсутствие кроссовера, этот алгоритм в действии очень похож на упомянутые выше генетические алгоритмы и поэтому легко сравним с ними. Это примечательно, потому что из этого демонстрируется эффектичность с точки зрения количества оценок целевой функции.

### Итоги сравнения

Ни один из вышеупомянутых многонишевых алгоритмов не хранит информацию обо всех оценённых ранее особях. Кумулятивный многонишевый генетический алгоритм заимствует идеи от многих вышеперечисленных генетических алгоритмов. В некоторых случаях заимствует определённые техники, но применяет их на иных стадиях генетического алгоритма. Комбинация генетических операций сделала функционирование алгоритма уникальным.

## Особенности реализации алгоритма

В процессе реализации алгоритма выявились некоторые особенности как генетических алгоритмов в целом, так и КМН ГА в частности. Подробнее о них ниже.

### Хранение генетического кода

Генетический код особей логически хранится в виде кода Грея. Двоичный код плохо подходит для этих целей, так как является позиционным: изменения в старших разрядах скажутся на особи сильнее, чем в младших. Код Грея лишён этого недостатка, и хотя это накладывает дополнительные расходы процессорного времени на декодирование, его использование вполне обосновано. Сложность декодирования гена оценивается как O(d\*l2), где d – количество измерений пространства, l – длина хромосомы. Декодирование и кодироване кода Грея представлено в листингах 1 и 2.

vector<genetic::point\_number>\* genetic::genetic\_code::decode(genetic\_code \*gcode) {

unsigned int code\_size = gcode->gray\_code->size();

chromosome\_length chromosome\_length = gcode->chromosomeLength;

unsigned int dimension = code\_size / chromosome\_length;

vector<point\_number> \*numbers = new vector<point\_number> (dimension, 0);

vector<gene> binary (chromosome\_length, 0);

for (unsigned int k = 0; k < dimension; ++k) {

point\_number number = 0;

unsigned int chromosome\_ptr = k \* chromosome\_length;

for (unsigned int digit\_ptr = 0; digit\_ptr < chromosome\_length; ++digit\_ptr) {

for (unsigned int shift = digit\_ptr; shift < chromosome\_length; ++shift) {

binary[shift] ^= (gcode->gray\_code)->at(chromosome\_ptr + shift - digit\_ptr);

}

number <<= 1;

number += binary.at(digit\_ptr);

}

numbers->at(k) = number;

binary.assign(chromosome\_length, 0);

}

return numbers;

}

Листинг 1 - Декодирование кода Грея

genetic::genetic\_code\* genetic::genetic\_code::encode(chromosome\_length chromosome\_length, vector<point\_number>\* numbers) {

unsigned int dimension = numbers->size();

unsigned int code\_size = chromosome\_length \* dimension;

vector<gene>\* gray\_codes = new vector<gene> (code\_size);

for (unsigned int k = 0; k < dimension; ++k) {

point\_number gray\_code = (\*numbers)[k] ^ ((\*numbers)[k] >> 1);

unsigned int chromosome\_ptr = k \* chromosome\_length;

for (unsigned int gene\_ptr = chromosome\_length - 1; gene\_ptr > 0 ; --gene\_ptr) {

(\*gray\_codes)[chromosome\_ptr + gene\_ptr] = (gene)(gray\_code & 0x1);

gray\_code = gray\_code >> 1;

}

(\*gray\_codes)[chromosome\_ptr] = (gene)(gray\_code & 0x1);

}

return new genetic\_code(chromosome\_length, gray\_codes);

}

Листинг 2 - Кодирование кода Грея

### Генерирование нормально распределённой случайной величины

В операции мутации КМН ГА участвует нормально распределённая случайная величина. Для её получения было использовано преобразование Бокса – Мюллера [2], позволяющее получить две нормально распределённых случайных величины из двух равномерно распределённых. Код метода в листинге 3.

double genetic::normal\_rand(double expectation\_value, double standart\_deviation) {

double u, v, s;

do {

u = (double)rand() / (RAND\_MAX + 1) \* 2 - 1;

v = (double)rand() / (RAND\_MAX + 1) \* 2 - 1;

s = u \* u + v \* v;

} while (s > 1 || fabs(s) < 1e-9);

double gaussian\_destribution = u \* sqrt((-2.0) \* log(s) / s);

return standart\_deviation \* gaussian\_destribution + expectation\_value;

}

Листинг 3 - Генерация нормально распределённой случайной величины

### Выбор медианы массива и числа определённого ранга

Некоторым операциям масштабирования приспособленности требуется выбирать значение элемента массива определённого ранга. Для этой цели был реализован алгоритм быстрого выбора (QuickSelect), имеющий среднюю сложность O(n) [3]. Это позволяет повысить скорость выполнения одного из самых узких мест, с точки зрения производительности, алгоритма. Код в листинге 4.

int partition(vector<double>\* v, int left, int right, int pivotIndex) {

double pivotValue = (\*v)[pivotIndex];

swap(v, pivotIndex, right);

int storeIndex = left;

for (int k = left; k < right; ++k) {

if ((\*v)[k] < pivotValue) {

swap(v, storeIndex, k);

++storeIndex;

}

}

swap(v, right, storeIndex);

return storeIndex;

}

double select\_rank(vector<double>\* v, int rank, int size) {

int left = 0;

int right = size - 1;

while ( true ) {

int pivotIndex = left + floor((double)rand() / (RAND\_MAX + 1) \* (right - left + 1));

pivotIndex = partition(v, left, right, pivotIndex);

if (rank == pivotIndex)

return (\*v)[rank];

else if (rank < pivotIndex)

right = pivotIndex - 1;

else

left = pivotIndex + 1;

}

}

Листинг 4 - Код алгоритма быстрого выбора

### Пропорционалный приспособленности отбор

Для выбора P1-родителя используется отбор пропорциональный приспособленности (FPS)при котором особи располагаются на рулетке в секторах, пропорциональных их приспособленности. Рулетка была реализована, как показано в листинге 5. Для поиска особи, на которую указала рулетка, использовался алгоритм бинарного поиска (листинг 6).

vector<int>\* P1 = new vector<int> (parameters->n\_crossover);

//-- P1: FPS --

cout << "P1 Selecting: Probability Calculating ... ";

vector<double>\* probability = new vector<double> (current\_population);

(\*probability)[0] = (\*fitness3)[0] / fitness3\_summ \* RAND\_MAX;

for (int k = 1; k < current\_population - 1; ++k) {

(\*probability)[k] = (\*fitness3)[k] / fitness3\_summ \* RAND\_MAX + (\*probability)[k - 1];

}

(\*probability)[current\_population - 1] = RAND\_MAX;

cout << "P1 Selecting: Select ... ";

for (int k = 0; k < parameters->n\_crossover; ++k) {

int rand\_value = rand();

int indIndex = binary\_search(probability, rand\_value);

(\*P1)[k] = indIndex;

}

delete probability;

Листинг 5 - Реализация рулетки FPS

int binary\_search(vector<double> \*v, int value) {

int left = 0;

int right = v->size() - 1;

while (left < right) {

int middle = left + (right - left) / 2;

if ((\*v)[middle] < value)

left = middle + 1;

else

right = middle - 1;

}

return left;

}

Листинг 6 - Бинарный поиск

## Результаты тестирования

Для сравнения с КМН ГА был выбран классический ГА как наиболее легко реализуемый. В тестироании участвовали три мультимодальных функции:

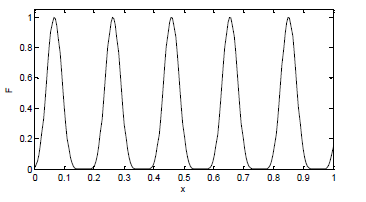


Рисунок 5 - График функции F1

, имеющая 5 одинаковых оптимумов.

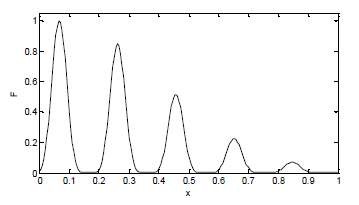


Рисунок 6 - График функции F2

, сконструирована так, чтобы оптимумы F1 стали разной высоты.

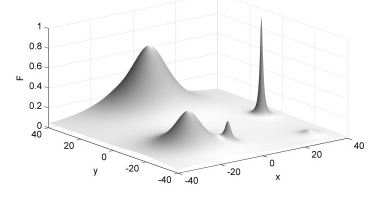
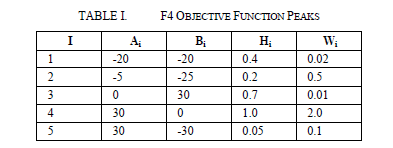


Рисунок 7 - График функции F2

, функция, имеющая 5 оптимумов, расположение, высота и ширина которых совпадает с соответствующими коэффициентами.



Поскольку генетические алгоритмы во многом полагаются на случайность, каждый алгоритм запускался несклько раз. В качестве итогового результата выбирался лучший.

Таблица 1 - Результаты работы алгоритмов для функции F1

|  |  |  |  |
| --- | --- | --- | --- |
| Алгоритм | Найденые оптимумы | Размер популяции | Время работы, с |
| Классический | Один и каждый раз разный | 10 | 0 |
| КМН ГА | 5 или более оптимумов | 23 | 0 |

Таблица 2 - Результаты работы алгоритмов для функции F2

|  |  |  |  |
| --- | --- | --- | --- |
| Алгоритм | Найденые оптимумы | Размер популяции | Время работы, с |
| Классический | Один и каждый раз разный | 10 | 0 |
| КМН ГА | 6 или более оптимумов | 45 | 1 |

Таблица 3 - Результаты работы алгоритмов для функции F3

|  |  |  |  |
| --- | --- | --- | --- |
| Алгоритм | Найденые оптимумы | Размер популяции | Время работы, с |
| Классический | Один и каждый раз разный | 10 | 0 |
| КМН ГА | 7 или более оптимумов | 274 | 0 |

## Заключение

* КМН ГА для сходимости требуется больший размер популяции. Это обусловлено природой алгоритма;
* при хорошо подобранных параметрах КМН ГА не пропускает локальные оптимумы и находит их все. Параметры можно подбирать, основываясь на предыдущих запусках алгоритма. Иногда находит несуществующие оптимумы;
* классический ГА часто сходится преждевременно, обладает нестабильным поведением на мультимодальных функциях;

## Приложение А. Часть листинга программы

**(genetic\_code.h)**

#pragma once

#include "function.h"

#include <vector>

#include <ctime>

#include <cmath>

using namespace std;

namespace genetic {

typedef unsigned \_\_int32 chromosome\_length;

typedef \_\_int32 point\_number;

typedef \_\_int8 gene;

double normal\_rand(double expectation\_value = 0, double standart\_deviation = 1);

class genetic\_code {

public:

vector<gene> \*gray\_code;

chromosome\_length chromosomeLength;

private:

genetic\_code(chromosome\_length chromosome\_length, vector<gene> \*genetic\_code);

public:

genetic\_code();

genetic\_code(int dimension, double \*coordinates);

~genetic\_code();

static void init\_rand();

static genetic\_code\* generate(chromosome\_length chromosome\_length, int dimension);

static genetic\_code\* encode(chromosome\_length chromosome\_length, vector<point\_number>\* numbers);

static vector<point\_number>\* decode(genetic\_code \*gcode);

static vector<double>\* decode(genetic\_code \*gcode, vector<bounds> \*bounds);

static double euclidean\_distance(genetic\_code \*gcode1, genetic\_code \*gcode2);

static genetic\_code\* crossover(genetic\_code\* parent\_1, genetic\_code\* parent\_2);

static genetic\_code\* mutate(genetic\_code\* mutant);

static genetic\_code\* mutate\_normal(genetic\_code\* mutant);

};

}

**(genetic\_code.cpp)**

#include "genetic\_code.h"

genetic::genetic\_code::genetic\_code() {

}

genetic::genetic\_code::genetic\_code(genetic::chromosome\_length chromosome\_length, vector<genetic::gene> \*genetic\_code) {

this->chromosomeLength = chromosome\_length;

this->gray\_code = new vector<gene> (\*genetic\_code);

}

genetic::genetic\_code::~genetic\_code() {

delete (this->gray\_code);

}

void genetic::genetic\_code::init\_rand() {

srand((unsigned) time( NULL ));

}

genetic::genetic\_code\* genetic::genetic\_code::generate(chromosome\_length chromosome\_length, int dimension) {

vector<gene> \*gcode = new vector<gene> (chromosome\_length \* dimension);

for (vector<gene>::iterator k = gcode->begin(); k < gcode->end(); ++k)

(\*k) = floor((double)rand() / RAND\_MAX + 0.5);

return new genetic\_code(chromosome\_length, gcode);

}

genetic::genetic\_code\* genetic::genetic\_code::encode(chromosome\_length chromosome\_length, vector<point\_number>\* numbers) {

unsigned int dimension = numbers->size();

unsigned int code\_size = chromosome\_length \* dimension;

vector<gene>\* gray\_codes = new vector<gene> (code\_size);

for (unsigned int k = 0; k < dimension; ++k) {

point\_number gray\_code = (\*numbers)[k] ^ ((\*numbers)[k] >> 1);

unsigned int chromosome\_ptr = k \* chromosome\_length;

for (unsigned int gene\_ptr = chromosome\_length - 1; gene\_ptr > 0 ; --gene\_ptr) {

(\*gray\_codes)[chromosome\_ptr + gene\_ptr] = (gene)(gray\_code & 0x1);

gray\_code = gray\_code >> 1;

}

(\*gray\_codes)[chromosome\_ptr] = (gene)(gray\_code & 0x1);

}

return new genetic\_code(chromosome\_length, gray\_codes);

}

vector<genetic::point\_number>\* genetic::genetic\_code::decode(genetic\_code \*gcode) {

unsigned int code\_size = gcode->gray\_code->size();

chromosome\_length chromosome\_length = gcode->chromosomeLength;

unsigned int dimension = code\_size / chromosome\_length;

vector<point\_number> \*numbers = new vector<point\_number> (dimension, 0);

vector<gene> binary (chromosome\_length, 0);

for (unsigned int k = 0; k < dimension; ++k) {

point\_number number = 0;

unsigned int chromosome\_ptr = k \* chromosome\_length;

for (unsigned int digit\_ptr = 0; digit\_ptr < chromosome\_length; ++digit\_ptr) {

for (unsigned int shift = digit\_ptr; shift < chromosome\_length; ++shift) {

binary[shift] ^= (gcode->gray\_code)->at(chromosome\_ptr + shift - digit\_ptr);

}

number <<= 1;

number += binary.at(digit\_ptr);

}

numbers->at(k) = number;

binary.assign(chromosome\_length, 0);

}

return numbers;

}

vector<double>\* genetic::genetic\_code::decode(genetic\_code \*gcode, vector<bounds> \*bounds) {

vector<point\_number>\* numbers = genetic\_code::decode(gcode);

unsigned int dimension = numbers->size();

int point\_count = 1 << gcode->chromosomeLength;

vector<double>\* coordinates = new vector<double> (dimension);

for (unsigned int k = 0; k < dimension; ++k) {

coordinates->at(k) = bounds->at(k).left + numbers->at(k) \* (bounds->at(k).right - bounds->at(k).left) / point\_count;

}

delete [] numbers;

return coordinates;

}

genetic::genetic\_code\* genetic::genetic\_code::crossover(genetic\_code\* parent\_1, genetic\_code\* parent\_2) {

//-- Размеры кодов должны совпадать!!! ---

unsigned int code\_size = parent\_1->gray\_code->size();

vector<gene>\* new\_code = new vector<gene> (code\_size);

for (int k = 0; k < code\_size; ++k) {

if (floor((double)rand() / RAND\_MAX + 0.5))

(\*new\_code)[k] = (\*parent\_1->gray\_code)[k];

else

(\*new\_code)[k] = (\*parent\_2->gray\_code)[k];

}

genetic\_code\* offspring = new genetic\_code(parent\_1->chromosomeLength, new\_code);

delete new\_code;

return offspring;

}

double genetic::normal\_rand(double expectation\_value, double standart\_deviation) {

double u, v, s;

do {

u = (double)rand() / (RAND\_MAX + 1) \* 2 - 1;

v = (double)rand() / (RAND\_MAX + 1) \* 2 - 1;

s = u \* u + v \* v;

} while (s > 1 || fabs(s) < 1e-9);

double gaussian\_destribution = u \* sqrt((-2.0) \* log(s) / s);

return standart\_deviation \* gaussian\_destribution + expectation\_value;

}

genetic::genetic\_code\* genetic::genetic\_code::mutate(genetic\_code\* mutant) {

int code\_size = mutant->gray\_code->size();

vector<gene> new\_code (code\_size);

for (int k = 0; k < code\_size; ++k) {

if ((double)normal\_rand() / RAND\_MAX < 0.5)

new\_code[k] = (\*mutant->gray\_code)[k];

else

new\_code[k] = !(\*mutant->gray\_code)[k];

}

return new genetic\_code(mutant->chromosomeLength, &new\_code);

}

genetic::genetic\_code\* genetic::genetic\_code::mutate\_normal(genetic\_code\* mutant) {

int dimension = mutant->gray\_code->size() / mutant->chromosomeLength;

int dimendions\_size = 1 << mutant->chromosomeLength;

//-- 40% design spase dimention--

double standart\_deviation = 0.4 \* dimendions\_size;

vector<point\_number>\* decoded = genetic\_code::decode(mutant);

vector<point\_number>\* mutate = new vector<point\_number> (dimension);

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

double real\_value = genetic::normal\_rand((double)(\*decoded)[k], standart\_deviation);

(\*mutate)[k] = floor(real\_value + 0.5);

}

genetic\_code\* encoded = genetic\_code::encode(mutant->chromosomeLength, mutate);

delete decoded;

delete mutate;

return encoded;

}

**(genetic\_algorithm.h)**

#pragma once

#include "function.h"

#include "genetic\_code.h"

#include <vector>

#include <ctime>

#include <cmath>

#include <iostream>

#include <string>

#include <new>

using namespace std;

using namespace genetic;

namespace genetic\_algorithm {

//-- параметры алгоритма --

struct parameters {

int n\_population;

int n\_crossover;

int n\_mutation;

int n\_min;

int n\_crowd;

int n\_try;

int n\_max\_population;

int n\_precision;

};

struct results {

vector<vector<double>\*>\* optimums;

unsigned int work\_time;

int final\_population\_size;

int final\_generation\_number;

};

results\* cmn\_ga(function::function func, vector<bounds>\* bounds, parameters\* parameters);

}

**(genetic\_algorithm.cpp)**

#pragma once

#include "genetic\_algorithm.h"

vector<double>\* decode(genetic\_code\* gcode, vector<double>\* left\_bounds, vector<double>\* step\_size, unsigned int dimension);

double euclidean\_distance(vector<double> \*v1, vector<double> \*v2);

double select\_rank(vector<double>\* v, int rank, int size);

double select\_median(vector<double>\* v, int size);

vector<int>\* nearest\_neighbours(vector<double>\* v, int n\_min, int size);

int binary\_search(vector<double> \*v, int value);

genetic\_algorithm::results\* genetic\_algorithm::cmn\_ga(function::function func, vector<bounds>\* bounds, parameters\* parameters) {

//-- общие параметны --

int t\_work;

int t\_start, t\_finish;

int generation\_number = 0;

//-- 3 знака после запятой --

int precision = parameters->n\_precision;

int max\_population = parameters->n\_max\_population;

unsigned int dimension = bounds->size();

vector<double> left\_bound (dimension);

vector<double> range\_size (dimension);

vector<double> step\_size (dimension);

double max\_range = 0;

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

left\_bound.at(k) = bounds->at(k).left;

range\_size.at(k) = bounds->at(k).right - bounds->at(k).left;

if (max\_range < range\_size.at(k))

max\_range = range\_size.at(k);

}

double stopping\_criterion = pow(10, precision);

chromosome\_length chromosome\_length = ceil(log(max\_range \* stopping\_criterion \* stopping\_criterion));

point\_number point\_count = 1 << chromosome\_length;

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

step\_size.at(k) = range\_size.at(k) / (point\_count - 1);

}

//-- --

int local\_optimims\_final\_count;

vector<int>\* local\_optimums\_final;

vector<vector<double>\*>\* local\_optimums\_final\_decoded;

vector<double>\* fitness0 = new vector<double> (max\_population);

vector<double>\* fitness1;

vector<double>\* fitness2;

vector<double>\* fitness3 = new vector<double> (max\_population);

double min\_fitness0 = 1e+100;

double max\_fitness0 = 0;

vector<vector<double>\*>\* decoded = new vector<vector<double>\*> (max\_population);

vector<vector<double>\*>\* invert\_distance = new vector<vector<double>\*> (max\_population);

t\_start = clock();

#pragma region //-- Step 0: (Initialization) --

cout << "Initialization...";

vector<genetic\_code\*>\* gcodes = new vector<genetic\_code\*> (max\_population);

int current\_population = parameters->n\_population;

for (int k = 0; k < current\_population; ++k) {

gcodes->at(k) = genetic\_code::generate(chromosome\_length, dimension);

decoded->at(k) = decode(gcodes->at(k), &left\_bound, &step\_size, dimension);

fitness0->at(k) = func(\*decoded->at(k));

if (min\_fitness0 > fitness0->at(k))

min\_fitness0 = fitness0->at(k);

}

for (int k = 0; k < current\_population; ++k)

(\*invert\_distance)[k] = new vector<double> (max\_population);

for (int from = 0; from < current\_population; ++from) {

for (int to = from + 1; to < current\_population; ++to) {

double euc\_distance = euclidean\_distance((\*decoded)[from], (\*decoded)[to]);

(\*(\*invert\_distance)[from])[to] = 1.0 / euc\_distance;

(\*(\*invert\_distance)[to])[from] = (\*(\*invert\_distance)[from])[to];

}

}

cout << " OK" << endl;

#pragma endregion

double dot9\_in\_gpower = 1;

bool population\_overflow = false;

while (true) {

#pragma region //-- Step 1: (Fitness Scaling) --

++generation\_number;

cout << endl << "Population has " << current\_population << " individuals" << endl;

cout << "Fitness Scaling: generation " << generation\_number << " ..." << endl;

int local\_optimum\_count = 0;

vector<int>\* local\_optimums = new vector<int> (max\_population);

cout << "Local optimums detection ...";

for (int k = 0; k < current\_population; ++k) {

vector<int>\* neighbours = nearest\_neighbours((\*invert\_distance)[k], parameters->n\_min, current\_population);

bool is\_local\_optimum = true;

for (vector<int>::iterator i = neighbours->begin(); is\_local\_optimum && i < neighbours->end(); ++i) {

if ((\*fitness0)[k] < (\*fitness0)[\*i])

is\_local\_optimum = false;

}

if (is\_local\_optimum)

(\*local\_optimums)[local\_optimum\_count++] = k;

delete neighbours;

}

cout << "Done. " << "Population has " << local\_optimum\_count << " optimums." << endl;

cout << "Fitness 3 Calculating ... ";

double fitness3\_summ = 0;

for (int ind = 0; ind < current\_population; ++ind) {

bool is\_local\_optimum = false;

for (int k = 0; k < local\_optimum\_count && !is\_local\_optimum; ++k)

is\_local\_optimum = (ind == (\*local\_optimums)[k]);

if (is\_local\_optimum) {

//-- Если локальный оптимум, то масштабирование не нужно. Просто 1 --

(\*fitness3)[ind] = 1.0;

}

else {

//-- Если не локальный оптимум, то масштабируем --

//cout << "Fitness 1 Calculation ... ";

fitness1 = new vector<double> (local\_optimum\_count);

int local\_optimums\_fitter\_count = 0;

for (int k = 0; k < local\_optimum\_count; ++k) {

if ((\*fitness0)[ind] < (\*fitness0)[(\*local\_optimums)[k]]) {

double diff\_fitness0 = (\*fitness0)[(\*local\_optimums)[k]] - min\_fitness0;

(\*fitness1)[k] = ((\*fitness0)[ind] - min\_fitness0) / diff\_fitness0;

}

else {

(\*fitness1)[k] == 1.0;

}

}

//cout << "Done" << endl;

//cout << "Median Selection ... ";

vector<double>\* fitness\_temp = new vector<double> (\*fitness1);

double median = select\_median(fitness\_temp, local\_optimum\_count);

delete fitness\_temp;

//cout << "Done" << endl;

double power = log(0.5) / log(median);

//cout << "The Power is " << power << endl;

//cout << "Fitness 2 Calculation ... ";

fitness2 = new vector<double> (local\_optimum\_count);

for (int k = 0; k < local\_optimum\_count; ++k) {

(\*fitness2)[k] = pow((\*fitness1)[k], power);

}

//cout << "Done" << endl;

double invert\_distance\_summ = 0;

double invert\_distance\_weighted\_summ = 0;

for (int k = 0; k < local\_optimum\_count; ++k) {

invert\_distance\_summ += (\*(\*invert\_distance)[ind])[(\*local\_optimums)[k]];

invert\_distance\_weighted\_summ += (\*(\*invert\_distance)[ind])[(\*local\_optimums)[k]] \* (\*fitness2)[k];

}

(\*fitness3)[ind] = invert\_distance\_weighted\_summ / invert\_distance\_summ;

delete fitness1;

delete fitness2;

}

fitness3\_summ += (\*fitness3)[ind];

}

cout << "Done" << endl;

cout << " OK" << endl;

#pragma endregion

#pragma region //-- Step 4: (New generation) --

cout << "Stopping Criterion: generation " << generation\_number << " ...";

//-- Stopping criterion --

bool is\_overall\_converged = true;

bool is\_local\_converged = false;

for (int optimum\_number = 0; optimum\_number < local\_optimum\_count && is\_overall\_converged; ++optimum\_number) {

for (int k = 0; k < current\_population && !is\_local\_converged; ++k) {

if ((\*(\*invert\_distance)[(\*local\_optimums)[optimum\_number]])[k] > stopping\_criterion)

is\_local\_converged = true;

}

is\_overall\_converged = is\_local\_converged && is\_overall\_converged;

}

//-- Алгоритм сошёлся --

if (is\_overall\_converged) {

local\_optimums\_final = local\_optimums;

local\_optimims\_final\_count = local\_optimum\_count;

break;

}

//-- Алгоритм достиг максимальной популяции --

if (population\_overflow) {

local\_optimums\_final = local\_optimums;

local\_optimims\_final\_count = local\_optimum\_count;

break;

}

delete local\_optimums;

cout << " OK" << endl;

#pragma endregion

dot9\_in\_gpower \*= 0.9;

#pragma region //-- Step 2: (Crossover) --

cout << "Crossover: generation " << generation\_number << " ... " << endl;

vector<int>\* P1 = new vector<int> (parameters->n\_crossover);

vector<int>\* P2 = new vector<int> (parameters->n\_crossover);

//-- P1: FPS --

cout << "P1 Selecting: Probability Calculating ... ";

vector<double>\* probability = new vector<double> (current\_population);

(\*probability)[0] = (\*fitness3)[0] / fitness3\_summ \* RAND\_MAX;

for (int k = 1; k < current\_population - 1; ++k) {

(\*probability)[k] = (\*fitness3)[k] / fitness3\_summ \* RAND\_MAX + (\*probability)[k - 1];

}

(\*probability)[current\_population - 1] = RAND\_MAX;

cout << "P1 Selecting: Select ... ";

for (int k = 0; k < parameters->n\_crossover; ++k) {

int rand\_value = rand();

int indIndex = binary\_search(probability, rand\_value);

(\*P1)[k] = indIndex;

}

delete probability;

//-- P2: PPS --

cout << "P2 Selecting ... ";

for (int k = 0; k < parameters->n\_crossover; ++k) {

int max\_proximitest = (double)rand() / (RAND\_MAX + 1) \* current\_population;

for (int i = 1; i < parameters->n\_crowd; ++i) {

int new\_crowd = (double)rand() / (RAND\_MAX + 1) \* current\_population;

if ((\*(\*invert\_distance)[(\*P1)[k]])[max\_proximitest] < (\*(\*invert\_distance)[(\*P1)[k]])[new\_crowd])

max\_proximitest = new\_crowd;

}

(\*P2)[k] = max\_proximitest;

}

cout << "Selecting Done" << endl;

//-- New Individuals --

//int try\_number = 0;

cout << "Crossover ... " << endl;

vector<bool>\* p\_used = new vector<bool>(parameters->n\_crossover, false);

int p\_used\_count = 0;

cout << "Try ";

for (int try\_number = 0; try\_number < parameters->n\_try && !population\_overflow && p\_used\_count < parameters->n\_crossover; ++try\_number) {

cout << try\_number << " ";

for (int p\_index = 0; p\_index < parameters->n\_crossover && try\_number < parameters->n\_try; ++p\_index) {

if ((\*p\_used)[p\_index])

continue;

genetic\_code \*crossover\_code = genetic\_code::crossover((\*gcodes)[(\*P1)[p\_index]], (\*gcodes)[(\*P2)[p\_index]]);

vector<double> \*crossover\_code\_decoded = decode(crossover\_code, &left\_bound, &step\_size, dimension);

vector<double> \*crossover\_code\_distance = new vector<double> (max\_population);

int min\_distance\_index = 0;

for (int to = 0; to < current\_population; ++to) {

(\*crossover\_code\_distance)[to] = euclidean\_distance(crossover\_code\_decoded, (\*decoded)[to]);

if ((\*crossover\_code\_distance)[min\_distance\_index] > (\*crossover\_code\_distance)[to])

min\_distance\_index = to;

}

double r\_min = 0.08 \* (1.001 - (\*fitness3)[min\_distance\_index] \* (1.0 - 0.5 \* dot9\_in\_gpower));

if (r\_min < (\*crossover\_code\_distance)[min\_distance\_index]) {

//-- Добавляем --

(\*gcodes)[current\_population] = crossover\_code;

(\*decoded)[current\_population] = crossover\_code\_decoded;

(\*fitness0)[current\_population] = func(\*crossover\_code\_decoded);

if (min\_fitness0 > (\*fitness0)[current\_population])

min\_fitness0 = (\*fitness0)[current\_population];

for (int k = 0; k < crossover\_code\_distance->size(); ++k) {

(\*crossover\_code\_distance)[k] = 1 / (\*crossover\_code\_distance)[k];

}

(\*invert\_distance)[current\_population] = crossover\_code\_distance;

for (int k = 0; k < current\_population; ++k)

(\*(\*invert\_distance)[k])[current\_population] = (\*crossover\_code\_distance)[k];

++current\_population;

(\*p\_used)[p\_index] = true;

++p\_used\_count;

if (current\_population == max\_population) {

population\_overflow = true;

break;

}

}

else {

//-- Слишком близко --

delete crossover\_code\_distance;

delete crossover\_code\_decoded;

delete crossover\_code;

}

}

}

cout << endl;

delete p\_used;

cout << " OK" << endl;

#pragma endregion

#pragma region //-- Step 3: (Mutation) --

cout << "Mutation: generation " << generation\_number << " ..." << endl;

cout << "Try ";

for (int try\_number = 0; try\_number < parameters->n\_mutation && !population\_overflow; ++try\_number) {

cout << try\_number << " ";

int mutant\_index = (double)rand() / (RAND\_MAX + 1) \* current\_population;

genetic\_code\* mutant\_code = genetic\_code::mutate\_normal((\*gcodes)[mutant\_index]);

vector<double> \*mutant\_code\_decoded = decode(mutant\_code, &left\_bound, &step\_size, dimension);

vector<double> \*mutant\_code\_distance = new vector<double> (max\_population);

int min\_distance\_index = 0;

for (int to = 0; to < current\_population; ++to) {

(\*mutant\_code\_distance)[to] = euclidean\_distance(mutant\_code\_decoded, (\*decoded)[to]);

if ((\*mutant\_code\_distance)[min\_distance\_index] > (\*mutant\_code\_distance)[to])

min\_distance\_index = to;

}

double r\_min = 0.08 \* (1.001 - (\*fitness3)[min\_distance\_index] \* (1.0 - 0.5 \* dot9\_in\_gpower));

if (r\_min < (\*mutant\_code\_distance)[min\_distance\_index]) {

//-- Добавляем --

(\*gcodes)[current\_population] = mutant\_code;

(\*decoded)[current\_population] = mutant\_code\_decoded;

(\*fitness0)[current\_population] = func(\*mutant\_code\_decoded);

if (min\_fitness0 > (\*fitness0)[current\_population])

min\_fitness0 = (\*fitness0)[current\_population];

for (int k = 0; k < mutant\_code\_distance->size(); ++k) {

(\*mutant\_code\_distance)[k] = 1 / (\*mutant\_code\_distance)[k];

}

(\*invert\_distance)[current\_population] = mutant\_code\_distance;

for (int k = 0; k < current\_population; ++k)

(\*(\*invert\_distance)[k])[current\_population] = (\*mutant\_code\_distance)[k];

++current\_population;

if (current\_population == max\_population) {

population\_overflow = true;

break;

}

}

else {

//-- Слишком близко --

delete mutant\_code\_distance;

delete mutant\_code\_decoded;

delete mutant\_code;

}

}

cout << " OK" << endl;

#pragma endregion

}

cout << " OK" << endl << "Finalazing ...";

local\_optimums\_final\_decoded = new vector<vector<double>\*> (local\_optimims\_final\_count);

for (int k = 0; k < local\_optimims\_final\_count; ++k) {

(\*local\_optimums\_final\_decoded)[k] = new vector<double> ( \*(\*decoded)[(\*local\_optimums\_final)[k]] );

}

//-- --

t\_finish = clock();

t\_work = (double)(t\_finish - t\_start) / CLOCKS\_PER\_SEC;

for (int k = 0; k < current\_population; ++k) {

delete (\*gcodes)[k];

delete (\*decoded)[k];

delete (\*invert\_distance)[k];

}

delete gcodes;

delete decoded;

delete fitness0;

delete fitness3;

delete invert\_distance;

delete local\_optimums\_final;

genetic\_algorithm::results\* results = new genetic\_algorithm::results();

results->optimums = local\_optimums\_final\_decoded;

results->work\_time = t\_work;

results->final\_population\_size = current\_population;

results->final\_generation\_number = generation\_number;

cout << " OK" << endl;

return results;

}

vector<double>\* decode(genetic\_code\* gcode, vector<double>\* left\_bounds, vector<double>\* step\_size, unsigned int dimension) {

vector<point\_number>\* numbers = genetic\_code::decode(gcode);

vector<double>\* coordinates = new vector<double> (dimension);

for (unsigned int k = 0; k < dimension; ++k) {

coordinates->at(k) = left\_bounds->at(k) + numbers->at(k) \* step\_size->at(k);

}

delete numbers;

return coordinates;

}

double euclidean\_distance(vector<double> \*v1, vector<double> \*v2) {

int dimension = v1->size();

double summ = 0;

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

double summand = (\*v1)[k] - (\*v2)[k];

summ += summand \* summand;

}

return sqrt(summ);

}

vector<int>\* nearest\_neighbours(vector<double>\* v, int n\_min, int size) {

int rank = size - n\_min - 1;

vector<double>\* temp\_v = new vector<double> (size);

for (int k = 0; k < size; ++k)

(\*temp\_v)[k] = (\*v)[k];

double rank\_value = select\_rank(temp\_v, rank, size);

delete temp\_v;

vector<int>\* nearest = new vector<int> (n\_min);

int index = 0;

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

if ((\*v)[k] > rank\_value) {

(\*nearest)[index++] = k;

}

}

return nearest;

}

#pragma region Quick Select

void swap(vector<double>\* v, int i, int j) {

double temp = v->at(i);

(\*v)[i] = (\*v)[j];

(\*v)[j] = temp;

}

int partition(vector<double>\* v, int left, int right, int pivotIndex) {

double pivotValue = (\*v)[pivotIndex];

swap(v, pivotIndex, right);

int storeIndex = left;

for (int k = left; k < right; ++k) {

if ((\*v)[k] < pivotValue) {

swap(v, storeIndex, k);

++storeIndex;

}

}

swap(v, right, storeIndex);

return storeIndex;

}

double select\_rank(vector<double>\* v, int rank, int size) {

int left = 0;

int right = size - 1;

while ( true ) {

int pivotIndex = left + floor((double)rand() / (RAND\_MAX + 1) \* (right - left + 1));

pivotIndex = partition(v, left, right, pivotIndex);

if (rank == pivotIndex)

return (\*v)[rank];

else if (rank < pivotIndex)

right = pivotIndex - 1;

else

left = pivotIndex + 1;

}

}

double select\_median(vector<double>\* v, int size) {

int median = ceil(size / 2.0) - 1;

return select\_rank(v, median, size);

}

#pragma endregion

int binary\_search(vector<double> \*v, int value) {

int left = 0;

int right = v->size() - 1;

while (left < right) {

int middle = left + (right - left) / 2;

if ((\*v)[middle] < value)

left = middle + 1;

else

right = middle - 1;

}

return left;

}

## Приложение Б. Перевод статьи

**Кумулятивный многонишевый генетический алгоритм   
для оптимизации мультимодальных функций**

Мэтью Холл  
Факультет машинной инжинерии  
Университета Виктории  
Виктория, Канада

**Конспект** – Эта статья представляет кумулятивный многонишевый генетический алгоритм, разработанный для быстрого решения задач оптимизации, имеющих высокую вычислительную сложность для мультимодальных целевых функций. Никогда не выбрасывая особей из популяции, CMN GA использует информацию от каждой оценки целевой функции как исследование конструкции пространства. Соответственно связанные плотности популяции управляют конструкцией пространства <design space>, сокращая необязательные (не необходимые) оценки целевой функции. Новый механизм генетических операций алгоритма обеспечивает быструю и устойчивую сходимость к множественным локальным экстремумам. Бенчмарки с другими тремя <multi-niching algorithms> показали, что CMN GA имеет лучшую сходимость и обеспечивает на порядок сниженное количество оценок функции, требуемое для достижения указанной скорости сходимости.

**Введение**

Генетические алгоритмы предоставляют мощную концептуальную основу для создания индивидуальных <customized> инструментов оптимизации, способных ориентироваться <navigate> в сложных прерывающихся <discontinuous> конструкциях пространства, ставящие в тупик иные методы. В этой статье представляется новый генетический алгоритм, который уникально <uniquely> соединяет два ключевых свойства: высокая эффективность в количестве оценок целевой функции, необходимых для достижения сходимости, и устойчивость при оптимизации мультимодальных целевых функций. Алгоритм был создан с этими свойствами для решения специфической задачи: разработка плавучих платформ для ветровых турбин <the design of floating platforms for offshore wind turbines>. Однако, свойства алгоритма делают его потенциально значимым для любых задач, особенностями которых являются высокая вычислительная ёмкость целевой функции и множество локальных оптимумов в прерывающихся конструкциях пространства.

Множество проектов задач оптимизации имеют целевые функции с высокой вычислительной ёмкостью. Не смотря на то, что генетические алгоритмы могут быть идеальными во многих отношениях, для классического генетического алгоритма избавление от оценённых особей из предыдущего поколения составляет излишнюю потерю информации. Вместо того, чтобы быть отброшенными эти особи могли бы быть сохранены и использованы для информирования алгоритма о хороших и плохих регионах в конструкции пространства и предотвращать избыточные оценки почти идентичных особей. Это может ускорить процесс оптимизации значительно снизив число оценок целевой функции, требуемых для сходимости к оптимальному решению.

В литературе примеры генетических алгоритмов, хранящих оценённых особей в памяти для снижения ненужных или излишних оценок целевой функции, редки. Зион и Шнайдер (Xiong and Schneider), что они называют <Cumulative GA>, который хранит всех особей с высоким значением приспособленности, для использования вместе с текущим поколением в размножении. Это достижение полезно для сохранения информации о лучшем регионе конструкции пространства, но это не позволяет избежать избыточных оценок целевой функции. Генетический алгоритм, разработанный Гантовником (Gantovnik), однако, позволяет. Его алгоритм хранит информацию обо всех предыдущих особях и использует её для построения метода Шепарда (Shepard) реакции поверхностной аппроксимации окружающих значений приспособленности (Shepard’s method response surface approximation of surrounding fitness values), который может быть использован вместо оценки целевой функции для близких особей.

Сохранение последних особей для обеих этих проблем предоставляет информацию о конструкции пространства и позволяет избежать излишних оценок целевой функции было первой целью в разработке нового генетического алгоритма. Второй целью была способность алгоритма определять и сходиться около множества локальных оптимумов одинаково (equitable).

Определение множества локальных оптимумов необходимо для множества практических задач оптимизации, имеющих мультимодальные целевые функции. Даже если целевая функция имеет только один глобальный оптимум, другой локальный оптимум может, в действительности, быть предпочтительным выбором по одному из дополнительных факторов, считающихся факторами, которые могут быть слишком сложными, качественными или субъективными чтобы быть включёнными в целевую функцию (Even though an objective function may have only one global optimum, another local optimum may in fact be the preferred choice once additional factors are considered – factors that may be too complex, qualitative, or subjective to be included in the objective function). Для примера, в оптимизации плавучих платформ для ветровых турбин число существующих отдельных локально-оптимальных конструкций, ранжируется от широких барж до глубоких стройных вех (deep slender spar-buoys). Хотя вехи могут иметь большую устойчивость (обычный выбор целевой функции), баржевый дизайн может быть предпочтительнее считая простоту установки.

Более того, в глобальной оптимизации часто используются грубые приближения целевой функции для ради скорости исследования структуры больших пространств. Для таких приближений возможно так искривить структуру пространства, что неверный локальный оптимум станет глобальным в оптимумом в приближении целевой функции. В таких случаях рекоммендуется в качестве второй стадии оптимизации использовать локальные, использующий градиент, методы оптимизации с более точной моделью целевой функции для проверки положения локальлных оптимумов и определения, который из них действительно является глобальным оптимумом.

Классический генетический алгоритм стабильно сходится только к одному локальному оптимуму, однако были разработаны подходы, позволяющий алгоритму сходиться к множеству локальных оптимумов. Такую способность называют «multi-niching». Подход разделения <Sharing>, предложенный Холландом [3] и раскрытый Голдбергом и Ричардсоном [4], снизил приспособленность каждой особи, основанной на количестве соседних особей. Снижение приспособленности определено функцией разделения <sharing function>, которая включает предельную дистанцию, определяющую какой уровень сходства составляют соседние особи. Недостатком этого метода является то, что для выбора хорошей функции разделения необходимо априори знать характеристики целевой функции. Также этот метод имеет трудности при формировании стабильной подпопуляции, хотя в этой области были сделаны улучшения [5].

Альтернативой является метод вытеснения/сгущения <Crowding> Де Ёнга [6], особенностья которого является замена шага, определяющего какие особи составят следующее поколение: для каждого потомка выбирается случайное подмножество особей из текущей популяции и особь из него, наиболее схожая с потомком, заменяется им. Улучшение, внесённое Махфоундом и названное детерминированное вытеснение <Deterministic crowding> [7], убирающее давление отбора в репродукции использованием случайного, а не пропорционального приспособленности отбора и изменившее шаг замены так, что каждый кроссоверный потомок конкурирует с более похожим на него родителем за вхождение в следующее поколение.

«Multi-Niche Crowding» метод Седено [8] отдичается от предыдущих методов вытеснения реализацией принципа выталкивания на стадии отбора. Для каждой кроссовер-пары один родитель, выбирается случайно или последовательно, а другой как наиболее схожая особь из группы случайно выбранных особей.

Это способствует спариванию между близлежащими особями, обеспечивая стабильность для multi-niching. Операция замены описана как «худший среди наиболее схожих»; количество групп создаётся случайно из популяции, особи из каждой группы, наиболее схожие с рассматриваемым потомком вбираются и наименее подходящий из этих «наиболее схожих» особей заменяется потомком.

Хотя метод «Multi-Niche» вытеснения вполне эффективен в поиске множества локальных оптимумов, он и другие методы, описанные выше, всё ещё отдают предпочтение оптимуму с наибольшим значением приспособленности. Ли, Чо и Юн приводят другой метод, названный Ограниченый конкурентный отбор <Restricted Competition Selection> [9], который превосходит предыдущие методы в нахождении и удержании даже слабого локального оптимума. В из ином-классическом <otherwise-conventional> методе каждая пара особей внутри «нишевого радиуса» <niche radius> друг от друга сравниваются и приспособленность менее приспособленной особи устанавливается авной нулю. Это в сущности оставляет только локально-оптимальных особей для репродукции. Самые приспособленные из тех особей сохраняются в следующем поколении, как элита <elites>.

Некоторые более поздние генетические алгоритмы используют информацию о направлении для для обеспечения лучшего управления в исследованием структуры пространства. Ху и др. <Hu et al.> зашли так далеко, что численно вычисляли градиент целевой функции для каждой особи для того, чтобы использовать метод наисколрейшего спуска <steepest descent> для выбора потомка [10].

Этот метод мощный, но большое количество оценок функции делает его неприменимым для вычислительно-ёмких целевых функций. Лиан и Леун [11] используют более умереный метод в котором два потенциальных потомка, создаваемые вместе линией, соединяющей две существующих особи и четыре результирующие значения приспособленности сравниваются для того, чтобы прогнозироватьположение соседних пиков. Используя эту информацию для информирования особенно устроенный <specially-constructed> кроссовер и мутацию, этот алгоритм использует значительно меньше оценок функции, чем другие участвующие в сравнении генетические алгоритмы [11].

Метод, использующий ещё меньше оценок функции, - эволюционный алгоритм <evolutionary algorithm (EA)> Коеваса и Гонцалеса (Cuevas and Gonźalez), который имитирует колелктивное поведение животных [12]. Этот алгоритм моделирует путь животных влекомых или отталкиваемых от доминантных особей и сохраняет в памяти набор наиболее приспособленных особей. Конкуренция между особями, которые находятся ближе определённого порога, также включена. Не смотря на отсутствие кроссовера, этот алгоритм в действии очень похож на упомянутые выше генетические алгоритмы и поэтому легко сравним с ними. Это примечательно, потому что из этого демонстрируется эффектичность с точки зрения количества оценок целевой функции.

Ни один из вышеупомянутых «multi-niching» алгоритмов не хранит информацию обо всех оченённых ранее особях; генетический алгоритм, сочетающий такой тип памяти и «multi-niching» - новое творение. В разработке такого алгоритма, о котором говорили как о «Cumulative Multi-Niching (CMN) GA», были подчерпнуты идеи и вдохновение от многих из вышеупомянутых методов. В некоторых случаях были воспроизведены определённые техники, но на других стадиях работы генетического алгоритма. Комбинация генетических операций сделала функционирование генетического алгоритма полностью уникальным.

**Описание алгоритма**

Отличительная особенность CMN GA – это кумулятивность. Каждое последующее поколение добавляется к общей популяции. С цельюминимизации оценок фунции, оценённые особи никогда не исключаются; даже неприспособленные особи имеют значение с сообщении алгоритму куда не нужно идти. Ключём, для того чтобы кумулятивные метод работал, является использование адаптивного ограничения близости, что предотвращает появление отпрысков, которые очень схожи с существующими особями, добавленными в популяцию. Используя пороговое расстояние, обратно пропорциональное приспособленности соседних особей, CMN GA поощряет сходимость вокруг перспективных регионов структуры пространства и поддерживает разреженную плотность популяции в низкоприспособленных регионах структуры пространства.

Фундаментальное отличие CMN GA от других генетических алгоритмов в том, что он включает число уникальных особенностей в генетических опрациях алгоритма, которые соединённые вместе (как приведено на Рисунке 2) для того, чтобы кумулятивный многонишевый метод работал. Отбор и кроссовер разработаны для поддержания стабильных подпопуляций около локальных оптимумов и обеспечить <drive> сходимость алгоритма. Операция мутации разработана для поощрения многообразия и исследования структуры пространства. Операция «добавление», которая занимает место операции замены классического генетического алгоритма, разработана для обеспечения использования накопившейся популяции особей, для того чтобы избежать излишних или ненужных оценок целевой функции и руководить генетическим алгоритмом для создания отпрысков в наиболее перспективных областях структуры пространства. Операция масштабирования приспособленности позволяет генетическому алгоритму обрабатывать локальные оптимумы равнозначно несмотря на возможные различия в приспособленности. Подробнее об этих операциях ниже.

**Отбор и кроссовер**

Процесс отбора и спаривания для кроссовера объединяет пропорциональный приспособленности отбор и схему вытеснения-вдохновения <crowding-inspired>, смещённая в сторону соседних особей. В то время как многонишевый метод Седено выбирает первого родителя случайно, а его пару как ближайшую, среди случайно выбранных групп, CMN GA использует и фактор приспособленности, и близости в его операции отбора.

Первый родитель, P1, каждой пары выбирается из популяции, используя стандартный пропорциональный приспособленности отбор (FPS) с выроятностью выбора пропорциональной приспособленности. Затем, для каждого P1, группа из Ncrowd кандидатов, пара выбирается, используя отбор, пропорциональный близости (PPS), где вероятность выбора определяется вероятностной функцией, описывающей, насколько близко кандидат P2 к P1 в структуре пространства. Наиболее простая вероятностная функция – инверсия расстояния по Евклиду:

(1)

где X – переменная-вектор длины n решения особи. Наиболее приспособленный из толпы потенциальных пар выбирается для спаривания с P1. Этот процесс повторяется для каждой особи, выбранной в качестве P1 – родителя для кроссовера.

Благодаря тому, что парой к особи является наиболее приспособленная из группы, в основном, соседних особей, поощряется спаривание между особями из одной ниши, хотя вероятностный отбор группы позволяет редкие спаривания с удалёнными особями, обеспечивая важную возможность кроссовера между нишами. Этот подход способствует стабильной многонишевости CMN GA и является основой кроссовер-сходимости популяции к локальным оптимумам.

В операции кроссовера переманная-значение решения отпрыска выбирается равномерно-случайно из гиперкуба, ограниченного переменными-значениями решения обоих родителей.

**Мутация**

Операция мутации происходит паралельно с операцией кроссовера. Мутационный отбор выполняется случайно и мутация переменной решения каждой особи базируется на нормальном распределении по начальному значению с перестраиваемым стандартным отклонением. Это даёт алгоритму способность широкого изучения структуры пространства. Хотя приспособленность особей неявно используется в операции мутации, последующая операция добавления делает более вероятным, что мутация произойдёт в более приспособленных регионах структуры пространства.

**Добавление**

Кумулятивная природа CMN GA исключает использование операции замены. Вместо этого, операция добавления добавляет отпрысков к постоянно расширяющейся популяции. Ограничение близости гарантирует, что алгоритм сходится к более приспособленным особям и уходит от менее приспособленных. Эта фильтрация, имеющая место перед оценкой приспособленности отпрысков, ключевая для кумулятивного подхода к популяции. Исключая отпрысков, которые чрезмерно схожи с существующими членами популяции, избегаются излишние оценки целевой функции.

Пороговое дистанция ограничения близости, Rmin, обратно зависит от приспособленности соседних существующих особей, Fnearest, как определено функцией пороговой дистанции. Простым примером этого будет:

(2)

Значения этой функции в пороговой дистанции будет 0.001 около наиболее приспособленных особей и 0.101 около наименее приспособленных особей, где дистанция нормализована границами структуры пространства, и приспособленность масштабирована к промежутку [0..1].

Такой подход для функции добавления позволяет потомкам быть очень близко к приспособленным особям и обеспечивает большую минимальную дистанцию для менее приспособленных особей. По существу, плотность популяции будет высокой в хороших и низкой в плохих регионах структуры пространства, как определено накопленными оценками целевой функции в течение работы генетического алгоритма. Карта плотности популяции, по существу, прописывает структуру пространства как прогресс алгоритма. Если структура пространства известна априори, использование сеточного исследования структуры прстранства может быть более эффективным, но без этого знания более адаптивный метод будет более практичным.

Для приспособления алгоритма к изменяющимся задачам в ходе оптимизации – вначале для исследования структуры пространства, а затем для сужения на локальные оптимумы – в функции пороговой дистанции могут быть сделаны изменения в зависимости от количества особей или номера поколения, G. Это может помочь предотвратить преждевременную сходимость, обеспечив нахождение всех локальных оптимумов. Функция пороговой дистанции, которую использовал автор, для получения результатов в этой статье:

(3)

**Масштабирование приспособленности**

Алгоритм, описанный до сих пор, потенциально может сойтись только к наиболее приспособленному локальному оптимуму. Последний компонент, разработанный для ршения этой задачи и обеспечения одинакового отношения ко всем значимым локальным оптимумам, является операция близостно-весового масштабирования приспособленноти. В большинстве генетических алгоритмов функция масштабирования применяется к значениям приспособленности популяции для масштабирования их в нормализованные границы, а также чтобы регулировать распределение приспособленности. Поостейший подход – линейно масштабировать значение приспособленности, F, в промежуток [0..1], так что наименее значение приспособленности спроецируется в F’=0 и наибольшее значение в F’=1:

(4)

Функция масштабирования также может быть использована для регулирования распределения приспособленности в ширину <across> промежутка значений приспособленности для того, чтобы, например, обеспечить внимание более или менее к умерено приспособленным особям. Это масштабироване может быть адаптировано к особенностям популяции. Для получения результатов, представленных здесь, использовалась вторая экспоненциальная масштабирующая функция для регулирования значений масштабированной приспособленности так, чтобы медиана равнялась 0.5:

(5)

Близостно-весовое масштабирование приспособленности, ключевой компонент CMN GA, добавляет дополнительную операцию масштабирования. Эта операция зависит от обнаружения локально-оптимальных особей в популяции. Использованный критерий, для простоты, что особь считается представляющей локальный оптимум, если она приспособленнее чем все ,ближайшие Nmin соседи. В операции близостно-весового масштабирования масштабирующие функции (4) и (5) применяются к популяции множество раз, каждый раз нормализуя результаты к приспособленности различных локальных оптимумов. Так, если было найдено m локальных оптимумов, каждая особь популяции будет иметь m масштабированных значений приспособленности. Эти масштабированные значения приспособленности F’’ затем комбинируются для каждой особи i относительно близости особи к каждому соответствующему локальному оптимуму j для получения значений финальной масштабированой приспособленности популяции:

(6)

Близость, Pi,j, может быть расчитана, как в (1). Такой прочесс даёт каждому локальному оптимуму одинаково-масштабированное значение приспособленности, как проиллюстрировано на фигуре 1.

**Резюме**

Фигура 2 описывает общую структуру CMN GA, показывая последовательность операций и как операция добавления отфильтровывает неинформативных потомков. Следующий раздел демострирует эффективность алгоритма в многонишевой сходимости с минимальным количеством оценок целевой функции.

**Скорость работы алгоритма**

Для определения скорости работы, CMN GA был протестирован с тремя другими многонишевыми алгоритмами на четырёх мультимодальных целевых функциях. Эти мультимодальные функции использовались многими разработчиками первых многонишевых алгоритмов [8].

Первыя, F1, одномерная функция, имеющая пять одинаковых пиков, показана на фигуре 3.

(7)

Вторая, F2, модификация F1, имеющая пики разной высоты, показана на фигуре 4.

(8)

Третья функция, F3, двумерная лисьенорья <Foxholes> функция Шекеля c 25 пиками неодинаковой высоты, расположенные в сетке как 16 обособленных частей, показана на фигуре 5.

(9)

Четвёртая функция, F4, нерегулярная функция с 5 пиками различной высоты и ширины, как показано в Таблице 1 и фигуре 6.

(10)

Для функций F3 и F4 Ai и Bi будут координатами каждого пика. Также для фунции F4 Hi и Wi являются высотой и шириной каждого пика. Этими четыремя функциями тестировалась многонишеввы способности алгоритма по различным направлениям.

## Приложение В. Оригинал статьи

A Cumulative Multi-Niching Genetic Algorithm for Multimodal Function Optimization

Matthew Hall

Department of Mechanical Engineering

University of Victoria

Victoria, Canada

***Abstract*—This paper presents a cumulative multi-niching genetic algorithm (CMN GA), designed to expedite optimization problems that have computationally-expensive multimodal objective functions. By never discarding individuals from the population, the CMN GA makes use of the information from every objective function evaluation as it explores the design space. A fitness-related population density control over the design space reduces unnecessary objective function evaluations. The algorithm’s novel arrangement of genetic operations provides fast and robust convergence to multiple local optima. Benchmark tests alongside three other multi-niching algorithms show that the CMN GA has greater convergence ability and provides an order-of-magnitude reduction in the number of objective function evaluations required to achieve a given level of convergence.**

***Keywords- genetic algorithm; cumulative; memory; multi-niching; multi-modal; optimization; metaheuristic.***

I. INTRODUCTION

Genetic algorithms provide a powerful conceptual framework for creating customized optimization tools able to navigate complex discontinuous design spaces that could confound other optimization techniques. In this paper, I present a new genetic algorithm that uniquely combines two key capabilities: high efficiency in the number of objective function evaluations needed to achieve convergence, and robustness in optimizing over multi-modal objective functions. I created the algorithm with these capabilities to meet the needs of a very specific optimization problem: the design of floating platforms for offshore wind turbines. However, the algorithm’s features make it potentially valuable for any application that features a computationally-expensive objective function and multiple local optima in a discontinuous design space.

Many design optimization problems have computationally-expensive objective functions. While genetic algorithms (GAs) may be ideal optimizers in many ways, a conventional GA’s disposal of previously-evaluated individuals from past generations constitutes an unnecessary loss of information. Rather than being discarded, these individuals could instead be retained and used to both inform the algorithm about good and bad regions of the design space and prevent the redundant evaluation of nearly-identical individuals. This could accelerate the optimization process by significantly reducing the number of objective function evaluations required for convergence to an optimal solution.

Examples in the literature of GA approaches that store previously-evaluated individuals in memory to reduce unnecessary or redundant objective function evaluations are sparse. Xiong and Schneider [1] developed what they refer to as a Cumulative GA, which retains all individuals with a high fitness value to use along with the current generation in reproduction. This approach is useful in retaining information about the best regions of the design space, but it does nothing to avoid redundant objective function evaluations. A GA developed by Gantovnik et al. [2], however, does. Their GA stores information about all previous individuals and uses it to construct a Shepard’s method response surface approximation of surrounding fitness values, which can be used instead of evaluating the objective function for nearby individuals.

Retaining past individuals to both provide information about the design space and avoid redundant objective function evaluations was my first goal in developing a new GA. My second goal was for the algorithm to be able to identify and converge around multiple local optima in an equitable way.

Identifying multiple local optima is necessary for many practical optimization problems that have multimodal objective functions. Even though an objective function may have only one global optimum, another local optimum may in fact be the preferred choice once additional factors are considered – factors that may be too complex, qualitative, or subjective to be included in the objective function. In the optimization of floating offshore wind turbine platforms, for example, a number of distinct locally-optimal designs exist, ranging from wide barges to deep slender spar-buoys. Though a spar-buoy may have the greatest stability (a common objective function choice), a barge design may be the better choice once ease of installation is considered.

Furthermore, global optimizations often use significant modelling approximations in the objective function for the sake of speed in exploring large design spaces. It is possible for such approximations to skew the design space such that the wrong local optimum is the global optimum in the approximated objective function. In those cases, local gradient-based optimizations with higher-fidelity models in the objective function are advisable as a second optimization stage to verify the locations of the local optima and determine which one of them is in fact the global optimum.

A conventional GA will only converge stably to one local optimum but a number of approaches have been developed for enabling convergence to multiple local optima, a capability referred to as “multi-niching”. The Sharing approach, proposed by Holland [3] and expanded by Goldberg and Richardson [4], reduces the fitness of each individual based on the number of neighbouring individuals. The fitness reduction is determined by a sharing function, which includes a threshold distance that determines what level of similarity constitutes a neighbouring individual. A weakness of this approach is that choosing a good sharing function requires a-priori knowledge of the objective function characteristics. As well, the approach has difficulty in forming stable sub-populations, though improvements have been made in this area [5].

An alternative is the Crowding approach of De Jong [6], which features a replacement step that determines which individuals will make up the next generation: for each offspring, a random subset of the existing population is selected and from it the individual most similar to the offspring is replaced by it. Mahfoud’s improvement, called Deterministic Crowding [7], removes the selection pressure in reproduction by using random rather than fitness-proportionate selection, and modifies the replacement step such that each crossover offspring competes against the more similar of its parents to decide which of the two enters the next generation.

The Multi-Niche Crowding approach of Cedeño [8] differs from the previous crowding approaches by implementing the crowding concept in the selection stage. For each crossover pair, one parent is selected randomly or sequentially and the other parent is selected as the most similar individual out of a group of randomly selected individuals.

This promotes mating between nearby individuals, providing stability for multi-niching. The replacement operation is described as “worst among most similar”; a number of groups are created randomly from the population, the individual from each group most similar to the offspring in question is selected, and the least fit of these "most similar" individuals is replaced by the offspring.

Though the Multi-Niche Crowding approach is quite effective at finding multiple local optima, it and the other approaches described above still provide preferential treatment to optima with greater fitness values. Lee, Cho, and Jung provide another approach, called Restricted Competition Selection [9], that outperforms the previously-mentioned techniques in finding and retaining even weak local optima. In their otherwise-conventional approach, each pair of individuals that are within a “niche radius” of each other are compared and the less fit individual’s fitness is set to zero. This in effect leaves only the locally-optimal individuals to reproduce. A set of the fittest of these individuals is retained in the next generation as elites.

Some more recent GAs add the use of directional information to provide greater control of the design space exploration. Hu et al. go so far as to numerically calculate the gradient of the objective function at each individual in order to use a steepest descent method to choose offspring [10].

This approach is powerful, but its large number of function evaluations makes it impractical for computationally-expensive objective functions. Liang and Leung [11] use a more restrained approach in which two potential offspring are created along a line connecting two existing individuals and the four resulting fitness values are compared in order to predict the locations of nearby peaks. By using this information to inform specially-constructed crossover and mutation operators, this algorithm uses significantly fewer function evaluations than other comparable GAs [11].

An approach shown to use even fewer function evaluations is an evolutionary algorithm (EA) by Cuevas and Gonźalez that mimics collective animal behaviour [12]. This algorithm models the way animals are attracted to or repelled from dominant individuals, and retains in memory a set of the fittest individuals. Competition between individuals that are within a threshold distance is also included. Notwithstanding the lack of a crossover function, this algorithm is quite similar in operation to many of the abovementioned GAs and is therefore easily compared with them. It is noteworthy because of its demonstrated efficiency in terms of number of objective function evaluations.

None of the abovementioned multi-niching algorithms retains information about all the previously-evaluated individuals; a GA that combines this sort of memory with multi-niching is a novel creation. In developing such an algorithm, which I refer to as the Cumulative Multi-Niching (CMN) GA, I drew ideas and inspiration from many of the abovementioned approaches. In some cases, I replicated specific techniques, but in different stages of the GA process. The combination of genetic operations to make up a functioning GA is entirely unique.

II. ALGORITHM DESCRIPTION

The most distinctive feature of the CMN GA is that it is cumulative. Each successive generation adds to the overall population. With the goal of minimizing function evaluations, evaluated individuals are never discarded; even unfit individuals are valuable in telling the algorithm where not to go. The key to making the cumulative approach work is the use of an adaptive proximity constraint that prevents offspring that are overly similar to existing individuals from being added to the population. By using a distance threshold that is inversely proportional to the fitness of nearby individuals, the CMN GA encourages convergence around promising regions of the design space and allows only a sparse population density in less-fit regions of the design space.

This fundamental difference from other GAs enables a number of unique features in the genetic operations of the algorithm that together combine (as summarized in Fig. 2) to make the cumulative multi-niching approach work. The selection and crossover operations are designed to support stable sub-populations around local optima and drive the algorithm’s convergence. The mutation operation is designed to encourage diversity and exploration of the design space. The “addition” operation, which takes the place of the replacement operation of a conventional GA, is designed to make use of the accumulated population of individuals in order to avoid redundant or unnecessary fitness function evaluation and guide the GA to produce offspring in the most promising regions of the design space. The fitness scaling operation makes the GA treat local optima equally despite potential differences in fitness. The details of these operations are as follows.

*A. Selection and Crossover*

The selection and pairing process for crossover combines fitness-proportionate selection with a crowding-inspired pairing scheme that is biased toward nearby individuals. Whereas Cedeño’s Multi-Niche Crowding approach selects the first parent randomly and selects its mate as the nearest of a randomly-selected group, the CMN GA combines factors of both fitness and proximity in its selection operation.

The first parent, P1, of each pair is selected from the population using standard fitness-proportionate selection (FPS) – with the probability of selection proportional to fitness. Then, for each P1, a crowd of Ncrowd candidate mates is selected using what could be called proximity-proportionate selection (PPS) - with the probability of selection determined by a proximity function describing how close each potential candidate mate, P2, is to P1 in the design space. The most basic proximity function is the inverse of the Euclidean distance:



where X is an individual’s decision variable vector, with length n. The fittest of the crowd of candidate mates is then selected to pair with P1. This process is repeated for each individual selected to be a P1 parent for crossover.

By having an individual mate with the fittest of a crowd of individuals that are mostly neighbours, mating between members of the same niche is encouraged, though the probability-based selection of the crowd allows occasional mating with distant individuals, providing the important possibility of crossover between niches. This approach contributes to the CMN GA’s multi-niching stability and is the basis for crossover-driven convergence of the population to local optima.

In the crossover operation, an offspring’s decision variable values are selected at uniform random from the hypercube bounded by the decision variable values of the two parents.

*B. Mutation*

The mutation operation occurs in parallel with the crossover operation. Mutation selection is done at random, and the mutation of the decision variables of each individual is based on a normal distribution about the original values with a tuneable standard deviation. This gives the algorithm the capability to widely explore the design space. Though individual fitness is not explicitly used in the mutation operation, the addition operation that follows makes it more likely that mutations will happen in fitter regions of the design space.

*C. Addition*

The cumulative nature of the CMN GA precludes the use of a replacement operation. Instead, an addition operation adds offspring to the ever-expanding population. A proximity constraint ensures that the algorithm converges toward fitter individuals and away from less fit individuals. This filtering, which takes place before the offspring’s fitnesses are evaluated, is crucial to the success of the cumulative population approach. By rejecting offspring that are overly similar to existing members of the population, redundant objective function evaluations are avoided.

The proximity constraint’s distance threshold, Rmin, is inversely related to the fitness of the nearest existing individual, Fnearest, as determined by a distance threshold function. A simple example is:



This function results in a distance threshold of 0.001 around the most fit individual and 0.101 around the least fit individual, where distance is normalized by the bounds of the design space and fitness is scaled to the range [0 1].

This approach for the addition function allows new offspring to be quite close to existing fit individuals but enforces a larger minimum distance around less fit individuals. As such, the population density is kept high in good regions and low in poor regions of the design space, as determined by the accumulated objective function evaluations over the course of the GA run. A population density map is essentially prescribed over the design space as the algorithm progresses. If the design space was known a priori, the use of a grid-type exploration of the design space could be more efficient, but without that knowledge, this more adaptive approach is more practical.

To adjust for the changing objectives of the algorithm as the optimization progresses – initially to explore the design space and later to narrow in on local optima - the distance threshold function can be made to change with the number of individuals or generation number, G. This can help prevent premature convergence, ensuring all local optima are identified. The distance threshold function that I used to generate the results in this paper is:



*D. Fitness Scaling*

The algorithm described thus far could potentially converge to only the fittest local optimum and not adequately explore other local optima. The final component, developed to resolve this problem and provide equitable treatment of all significant local optima, is a proximity-weighted fitness scaling operation. In most GAs, a scaling function is applied to the population’s fitness values to scale them to within normalized bounds and also sometimes to adjust the fitness distribution. A basic approach is to linearly scale the fitness values, F, to the range [0, 1] so that the least fit individual gets a scaled fitness of F’=0 and the fittest individual gets a scaled fitness of F’=1:



A scaling function can also be used to adjust the distribution of fitness across the range of fitness values in order to, for example, provide more or less emphasis on moderately-fit individuals. This scaling can be adaptive to the characteristics of the population. For the results presented here, I used a second, exponential scaling function to adjust the scaled fitness values so that the median value is 0.5:



Proximity-weighted fitness scaling, a key component of the CMN GA, adds an additional scaling operation. This operation relies on the detection of locally-optimal individuals in the population. The criterion I used, for simplicity, is that an individual is considered to represent a local optimum if it is fitter than all of its nearest Nmin neighbours. In the proximity-weighted fitness scaling operation, scaling functions (4) and (5) are applied multiple times to the population, each time normalizing the results to the fitness of a different local optimum. So if m local optima have been identified, each individual in the population will have m scaled fitness values. These scaled fitness values F’’ are then combined for each individual i according to the individual’s proximity to each respective local optimum j to obtain the population’s final scaled fitness values:



Proximity, Pi,j, can be calculated as in (1). This process gives each local optimum an equal scaled fitness value, as is illustrated for a one-dimensional objective function in Fig. 1.

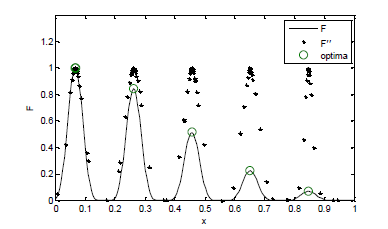


Figure 1. Proximity-weighted fitness scaling.

E. CMN GA Summary

Fig. 2 describes the overall structure of the CMN GA, outlining how the algorithm’s operations are ordered and how the addition operation filters out uninformative offspring. The next section demonstrates the algorithm’s effectiveness at multi-niche convergence with a minimal number of objective function evaluations.

III. PERFORMANCE RESULTS

To benchmark the CMN GA’s performance, I tested it alongside three other multi-niching algorithms on four generic multimodal objective functions. These four multimodal functions have been used by many of the original developers of multi-niching GAs [8].

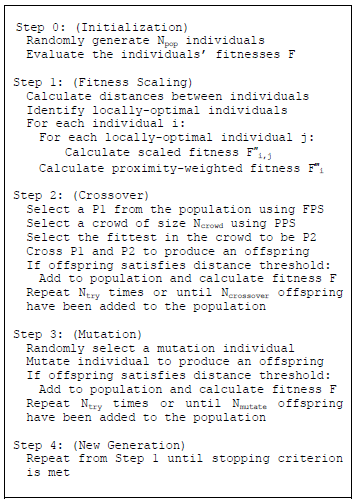


Figure 2. CMN GA outline.

The first, F1, is a one-dimensional function featuring five equal peaks, shown in Fig. 3.



The second, F2, modifies F1 to have peaks of different heights, shown in Fig. 4.



The third, F3, is a two-dimensional Shekel Foxholes function with 25 peaks of unequal height, spaced 16 units apart in a grid, as shown in Fig. 5.



The fourth, F4, is an irregular function with five peaks of different heights and widths, as listed in Table 1 and shown in Fig. 6.



In F3 (9) and F4 (10), Ai and Bi are the x and y coordinates of each peak. In F4 (10), Hi and Wi are the height and width parameters for each peak. These four functions test the algorithms’ multi-niching capabilities in different ways.

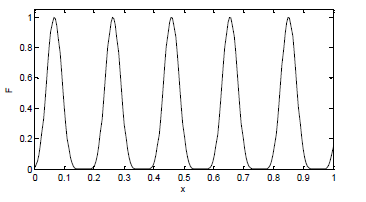


Figure 3. F1 objective function.

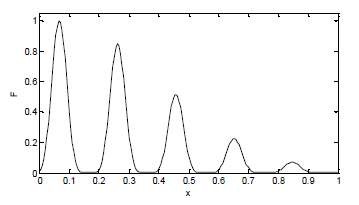


Figure 4. F2 objective function.

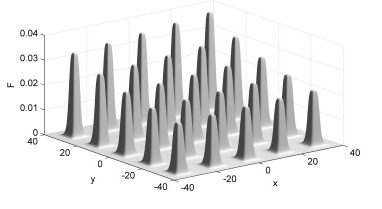


Figure 5. F3 objective function.

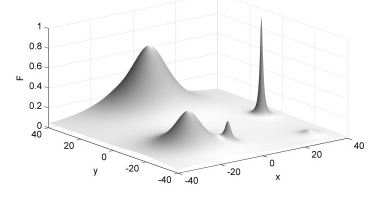
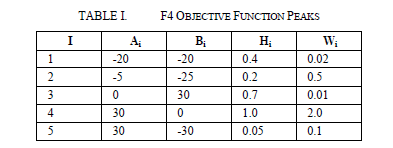


Figure 6. F4 objective function.



The two other multi-niching GA approaches I compare the CMN GA against are Multi-Niche Crowding (MNC) [8] and Restricted Competition Selection (RCS) [9]. I chose these two because they are very well-performing examples of two different approaches to GA multi-niching. I implemented these techniques into a GA framework that is otherwise the same as the CMN GA in terms of how it performs the crossover and mutation operations.

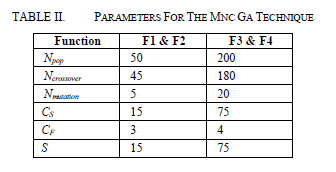
Crossover offspring decision variable values are chosen at uniform random from the intervals between the decision variables of the two parents. Mutation offspring decision variable are chosen at random using normal distributions about the unmutated values with standard deviations of 40% of the design space dimensions.

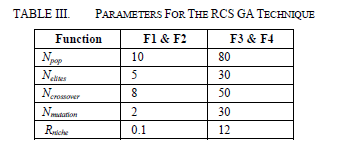
For further comparison, I also implemented the Collective Animal Behaviour (CAB) evolutionary algorithm [12]. It is a good comparator because it has many common features with multi-niching GAs, but has been shown to give better performance than many of them, particularly in terms of objective function evaluation requirements.

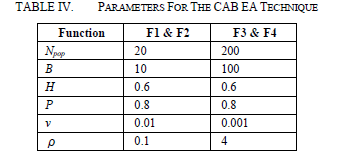
The values of the key tunable parameters used in each algorithm are given in Tables 2 to 5. Npop describes the population size, or the initial population size in the case of the CMN GA. For the RCS GA, Nelites is the number of individuals that are preserved in the next generation. I tuned the parameter values heuristically for best performance on the objective functions. For the MNC, RCS, and CAB algorithms, I began by using the values from [8], [9], and [12], respectively, but found that modification of some parameters gave better results. The meanings of the variables in Table 4 can be found in [12].

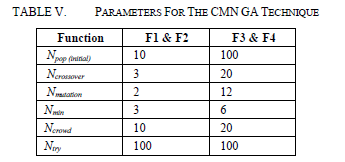
To account for the randomness inherent in the operation of a genetic or evolutionary algorithm, I ran each algorithm ten times on each objective function to obtain a reliable characterization of performance. The metric I use to measure the convergence of the algorithms to the local optima is the sum of the distances from each local optimum X\*j to the nearest individual.

By indicating how close the algorithm is to identifying all of the true local optima, this aggregated metric represents what is of greatest interest in multimodal optimization applications. The assumption is that in real applications it will be trivial to determine which evaluated individuals represent local optima without a-priori knowledge of the objective function.









Figures 7 to 10 show plots of the convergence metric versus the number of objective function evaluations for each optimization run. Using these axes gives an indication of algorithm performance in terms of my two objectives for the CMN GA, convergence to multiple local optima and minimal objective function evaluations. Figures 7, 8, 9, and 10 compare the performance of each algorithm for objective functions F1, F2, F3, and F4, respectively.

In the results for objective function F4, the MNC and CAB algorithms consistently failed to identify the shallowest peak. Accordingly, I excluded this peak from the convergence metric calculations for these algorithms in the data of Fig. 10 in order to provide a more reasonable view of these algorithms’ performance. The CMN GA also missed this peak in one of the runs, as can by the one anomalous curve in Fig. 10, wherein the convergence metric stagnates at a value of 2. As is the case with other multi-niching algorithms, missing subtle local optima is a weakness of the CMN GA, but it can be mitigated

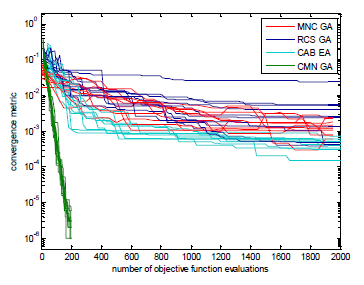


Figure 7. GA performance for F1 objective function runs.

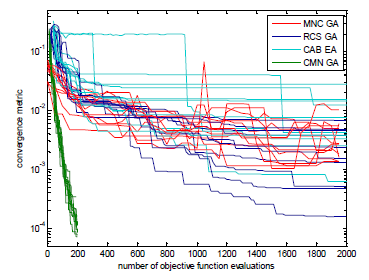


Figure 8. GA performance for F2 objective function runs.

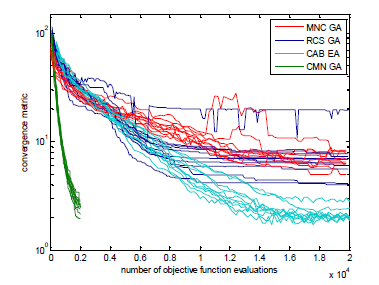


Figure 9. GA performance for F3 objective function runs.

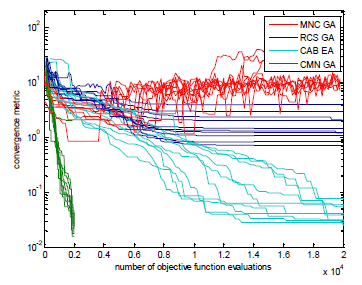


Figure 10. GA performance for F4 objective function runs.

by careful choice of algorithm parameters and verifying results through multiple optimization runs.

Fig. 11 is a snapshot of a population generated by the CMN GA on the F4 objective function. The distribution of the 1000 individuals in the figure illustrates how the algorithm clearly identifies the five local optima and produces a high population density around them regardless of how shallow or sharp they may be. Fig 12 shows how, with the same input parameters, the CMN GA is just as effective with the 25 local optima of the F3 objective function.

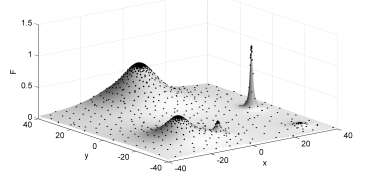


Figure 11. CMN GA exploration of F4 objective function.

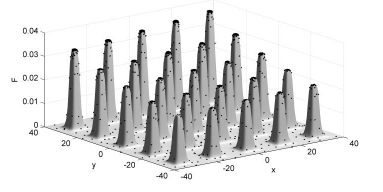


Figure 12. CMN GA exploration of F3 objective function.

Though more rigorous tuning of parameters could result in slight performance improvements in any of the four algorithms I compared, the order-of-magnitude faster convergence of the CMN GA gives strong evidence of its superior performance in terms of multimodal convergence versus number of objective function evaluations.

It should be noted that this measure of performance, reflective of the design goals of the CMN GA, is only indicative of performance on optimization problems where evaluating the objective function dominates the computational effort. The algorithm operations of the CMN GA are themselves much slower than those of the other algorithms, so the CMN GA could be inferior in terms of computation time on problems with easily-computed objective functions. As well, with its ever-growing population, the CMN GA’s memory requirements are greater than those of the other algorithms. In a sense, my choice of measure of performance puts the MNC, RCS, and CAB algorithms at a disadvantage because, unlike the CMN GA, these algorithms were not designed specifically for computationally-intensive objective functions. That said, convergence versus number of function evaluations is the most relevant measure of performance for optimizing over computationally-expensive multimodal objective functions, and the algorithms I chose for comparison represent three of the best existing options out of the selection of applicable GA/EA approaches available in the literature.

IV. CONCLUSION

In the interest of efficiently finding local optima in computationally-expensive objective functions, I created a genetic algorithm that converges robustly to multiple local optima with a comparatively small number of objective function evaluations. It does so using a novel arrangement of genetic operations in which new individuals are continuously added to the population; I therefore call it a Cumulative Multi- Niching Genetic Algorithm. The tests presented in this paper demonstrate that the CMN GA meets its goals – convergence to multiple local optima with minimal objective function evaluations – strikingly better than alternative genetic or evolutionary algorithms available in the literature. It therefore represents a useful new capability for optimization problems that have computationally-expensive multimodal objective functions. The proximity constraint approach used to control the accumulation of individuals in the population may also be applicable to other metaheuristic algorithms.

REFERENCES

[1] Y. Xiong and J. B. Schneider, “Transportation network design using a cumulative genetic algorithm and neural network,” Transportation Research Record, no. 1364, 1992.

[2] V. B. Gantovnik, C. M. Anderson-Cook, Z. Gürdal, and L. T. Watson, “A genetic algorithm with memory for mixed discrete–continuous design optimization,” Computers & Structures, vol. 81, no. 20, pp. 2003–2009, Aug. 2003.

[3] J. H. Holland, Adaptation in natural and artificial systems: An introductory analysis with applications to biology, control, and artificial intelligence. U Michigan Press, 1975.

[4] D. E. Goldberg and J. Richardson, “Genetic algorithms with sharing for multimodal function optimization,” in Proceedings of the Second International Conference on Genetic Algorithms and their Application, 1987, pp. 41–49.

[5] B. L. Miller and M. J. Shaw, “Genetic algorithms with dynamic niche sharing for multimodal function optimization,” in Proceedings of IEEE International Conference on Evolutionary Computation, 1996, pp. 786–791.

[6] K. A. De Jong, “Analysis of the behavior of a class of genetic adaptive systems,” PhD Thesis, University of Michigan, 1975.

[7] S. W. Mahfoud, “Crowding and preselection revisited,” Parallel problem solving from nature, vol. 2, pp. 27–36, 1992.

[8] W. Cedeño, “The multi-niche crowding genetic algorithm: analysis and applications,” PhD Thesis, University of California Davis, 1995.

[9] C.-G. Lee, D.-H. Cho, and H.-K. Jung, “Niching genetic algorithm with restricted competition selection for multimodal function optimization,”

Magnetics, IEEE Transactions on, vol. 35, no. 3, pp. 1722 –1725, May 1999.

[10] Z. Hu, Z. Yi, L. Chao, and H. Jun, “Study on a novel crowding niche genetic algorithm,” in 2011 IEEE 2nd International Conference on Computing, Control and Industrial Engineering (CCIE), 2011, vol. 1, pp. 238 –241.

[11] Y. Liang and K.-S. Leung, “Genetic Algorithm with adaptive elitist-population strategies for multimodal function optimization,” Applied Soft Computing, vol. 11, no. 2, pp. 2017–2034, Mar. 2011.

[12] E. Cuevas and M. González, “An optimization algorithm for multimodal functions inspired by collective animal behavior,” Soft Computing, Sep. 2012.

## Приложение Д. Библиографический список

1. *(IJARAI) International Journal of Advanced Research in Artificial Intelligence, Vol. 1, No. 9, 2012;*
2. <http://ru.wikipedia.org/wiki/Преобразование_Бокса_—_Мюллера>
3. <http://en.wikipedia.org/wiki/Quickselect>