МИНОБРНАУКИ РОССИИ

Федеральное государственное бюджетное образовательное учреждение
высшего образования
«МИРЭА – Российский технологический университет»

РТУ МИРЭА

Институт информационных технологий
Кафедра вычислительной техники

КУРСОВАЯ РАБОТА

по дисциплине «Системный анализ данных в системах поддержки принятия решений»
(наименование дисциплины)

Тема курсовой работы «Музыкальная индустрия»

Студент группы ИКБО-04-22 Егоров Л.А. _____
(учебная группа, фамилия, имя отчество, студента) (подпись студента)

Руководитель курсовой работы к.т.н., доцент Сорокин А.Б. _____
(должность, звание, ученая степень) (подпись руководителя)

Рецензент (при наличии) Жегалин Л.Н. _____
(должность, звание, ученая степень) (подпись рецензента)

Работа представлена к защите «27» 12 2024г.

Допущен к защите «27» 12 2024г. отл.

Москва 2024 г.

МИНОБРНАУКИ РОССИИ

Федеральное государственное бюджетное образовательное учреждение
высшего образования
«МИРЭА – Российский технологический университет»

РТУ МИРЭА

Институт информационных технологий
Кафедра вычислительной техники

Утверждаю

Заведующий кафедрой _____
Подпись

Платонова О.В.
ФИО

«15» сентябрь 2024г.

ЗАДАНИЕ

на выполнение курсовой работы по дисциплине

«Системный анализ данных в системах поддержки принятия решений»

Студент Егоров Л.А. Группа ИКБО-04-22

Тема: «Музыкальная индустрия»

Исходные данные: Набор данных Pima Indians Diabetes Database, программа Protege, библиотеки scikit-learn, pandas, numpy, matplotlib.

Перечень вопросов, подлежащих разработке, и обязательного графического материала: реализовать консольное приложение: 1. Для онтологии по предметной области, 2. Алгоритма имитации отжига, 3. Алгоритма роя частиц, 4. Алгоритма пчелиной колонии, 5. Муравьиный алгоритм.

Срок представления к защите курсовой работы: **до «28» декабрь 2024г.**

Задание на курсовую работу выдал _____ (Сорокин А.Б.)
Подпись руководителя *Ф.И.О. руководителя*

Задание на курсовую работу получил «15» сентябрь 2024г

_____ (Егоров Л.А.)
Подпись обучающегося Ф.И.О. исполнителя

Москва 2024г.

ОТЗЫВ

на курсовую работу

по дисциплине «Системный анализ данных в системах поддержки принятия решений»

Студент Егоров Л.А. (ФИО студента) ИКБО-04-22 (Группа)

Характеристика курсовой работы

Критерий	Да	Нет	Не полностью
1. Соответствие содержания курсовой работы указанной теме	+		
2. Соответствие курсовой работы заданию	+		
3. Соответствие рекомендациям по оформлению текста, таблиц, рисунков и пр.	+		
4. Полнота выполнения всех пунктов задания	+		
5. Логичность и системность содержания курсовой работы	+		
6. Отсутствие фактических грубых ошибок	+		

Замечаний: нет

Рекомендуемая оценка: отлично

Сорокин А.Б.

Подпись руководителя ФИО руководителя

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

В современном мире, где объем информации стремительно растет, а системы поддержки принятия решений становятся неотъемлемой частью деятельности организаций и предприятий, возникает необходимость в эффективных методах анализа данных и оптимизации процессов.

Системный анализ данных является ключевым направлением, объединяющим гуманитарные и формальные методы познания. Современные подходы в этой области опираются на интеграцию концепций философии, математики и кибернетики, позволяя моделировать сложные системы и анализировать их поведение. Одним из таких инструментов является использование онтологий — представлений о моделях мира, которые обеспечивают структуризацию информации, упрощая её анализ и обработку.

В рамках данной курсовой работы была поставлена задача разработки и реализации онтологии «Музыкальная индустрия», а также исследования и применения различных алгоритмов оптимизации для решения задач в этой области. Онтология «Музыкальная индустрия» представляет собой структурированное описание ключевых понятий и отношений в музыкальной индустрии, что позволяет более эффективно анализировать и обрабатывать данные, связанные с этой сферой.

Вместе с онтологиями в системах анализа данных и поддержки принятия решений активно применяются алгоритмы оптимизации, такие как имитация отжига, алгоритм роя частиц, муравьиный, пчелиный и электромагнитный алгоритмы. Эти методы основаны на принципах самоорганизации и коллективного поведения, которые наблюдаются в природе. Использование таких подходов позволяет моделировать сложные системы и находить оптимальные решения в многомерных пространствах.

1 ОНТОЛОГИЯ

Возникновение онтологий и их стремительное развитие связано с проявлением в нашей реальности следующих новых факторов:

- колоссальный рост объемов информации, предъявляемых для обработки (анализа, использования) специалистам самых различных областей деятельности;
- чрезвычайная зашумленность этих потоков (повторы, противоречивость, разноуровневость, и т.п.);
- острая необходимость в использовании одних и тех же знаний разными специалистами в разных целях;
- всеобщая интернетизация нашей жизни и острая необходимость в структуризации информации для её представления пользователям и более эффективного поиска;
- необходимость сокращения времени на поиск нужной информации и повышения качества информационных услуг в Интернете.

Онтологии — это базы знаний специального типа, которые могут читаться и пониматься, отчуждаться от разработчика и/или физически разделяться их пользователями.

Существует много видов онтологий, однако одним из самых широко применяемых видов являются онтологии предметных областей, содержащие понятия определённой области знаний или входящих в неё областей. Формальная модель онтологии представлена следующей формулой:

$$O =< X, R, F > \ ,$$

где X — конечное множество концептов (понятий, терминов) предметной области, которую представляет онтология;

R — конечное множество отношений между концептами (понятиями, терминами) заданной предметной области;

F — конечное множество функций интерпретации (аксиоматизации), заданных на концептах и/или отношениях онтологии.

Существует много видов онтологий, однако одним из самых широко применяемых видов являются онтологии предметных областей, содержащие понятия определённой области знаний или входящих в неё областей.

1.1 Постановка задачи

Необходимо разработать онтологию выбранной предметной области — «Музыкальная индустрия». Данная предметная область выбрана в связи с тем, что в современном мире прослушивание музыки стало очень доступным с использованием стриминговых сервисов, и поэтому есть острая необходимость в систематизации музыки, чтобы её было доступно выкладывать в Интернет [3].

1.2 Описание онтологии

Основным продуктом звукозаписывающих компаний являются музыкальные записи — наиболее распространёнными из них являются песни и альбомы, являющиеся сборниками песен. Авторами альбомов выступают либо группы, либо отдельные музыканты, и обе эти категории также связаны между собой — группы состоят из музыкантов [1].

На основе этого описания можно составить онтологию, состоящую из следующих классов:

- «Музыкальная индустрия» — общий базовый класс для всех классов;
- «Музыкальная запись» — базовый класс для разных видов музыкальных записей, содержит общий слот «Название»;
- «Альбом» — класс для описания альбомов, содержит слоты «Год выхода» и «Исполнитель», ссылающийся на экземпляр класса «Исполнитель»;
- «Песня» — класс для описания песен, содержит слот «Входит в альбом», ссылающийся на экземпляр класса «Альбом»;

- «Исполнитель» — базовый класс для всех видов исполнителей, содержит общие слоты «Имя» и «Страна происхождения»;
- «Группа» — класс для описания групп, не содержит своих слотов;
- «Музыкант» — класс для описания музыкантов, содержит слот «Входит в группу», ссылающийся на экземпляр класса «Группа».

Данное описание использовано для построения графической схемы онтологии (Рисунок 1.2.1).

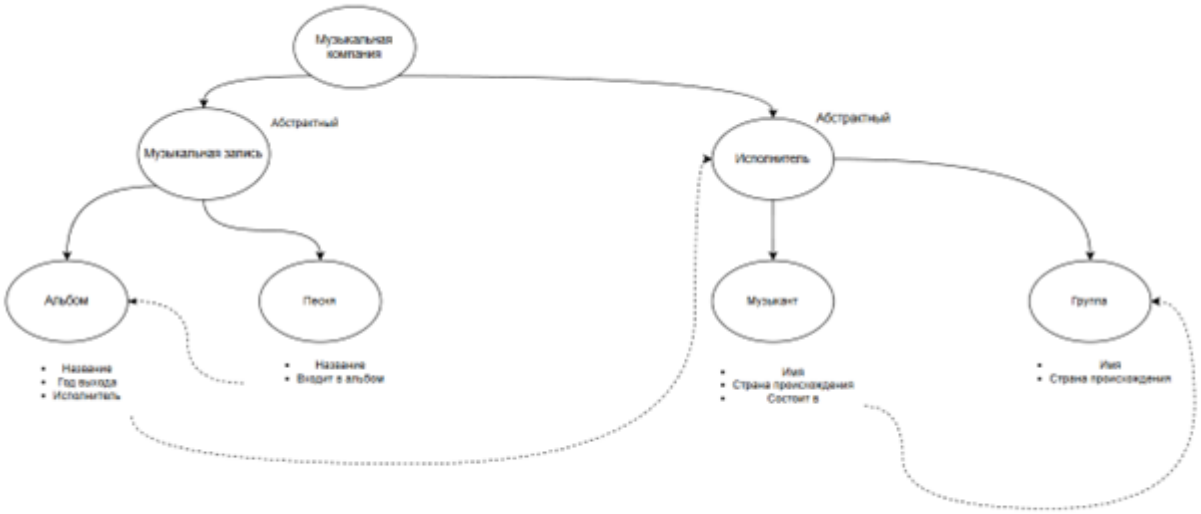

Рисунок 1.2.1 — Схема онтологии «Музыкальная индустрия»

1.3 Построение онтологии в Protégé

Для подробного изучения составленной онтологии использован инструмент для построения, редактирования онтологий и работы с ними Protégé. Сначала созданы классы, представленные на Рисунке 1.3.1.

Рисунок 1.3.1 — Составленная иерархия классов

На Рисунке 1.3.2 представлены слоты класса «Альбом». Слотами данного класса являются: название альбома, исполнитель и год выпуска.

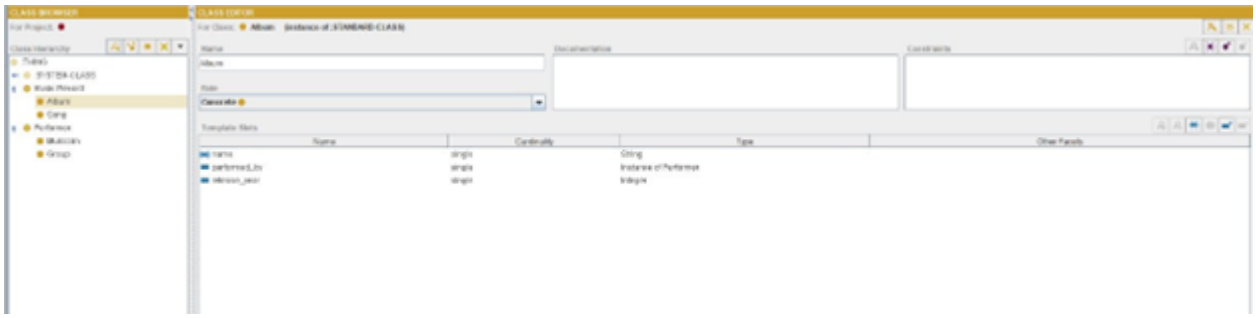

Рисунок 1.3.2 — Слоты класса «Альбом»

На Рисунке 1.3.3 представлены слоты класса «Песня». Слотами данного класса являются: альбом и название.

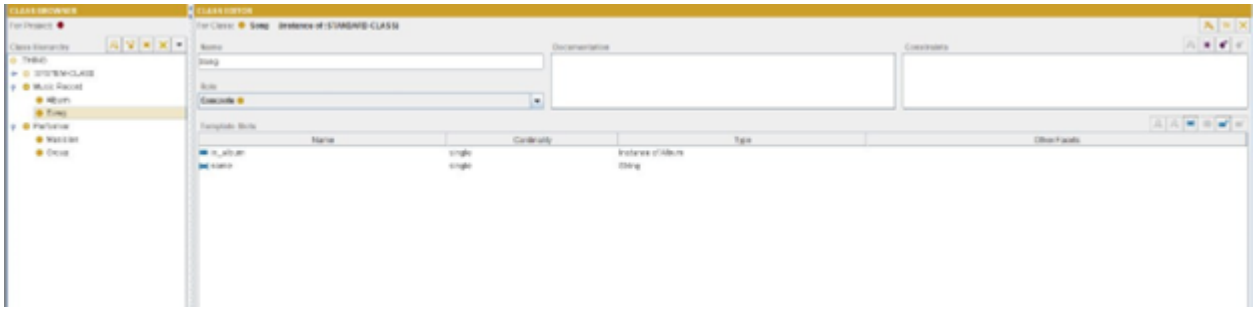

Рисунок 1.3.3 — Слоты класса «Песня»

На Рисунке 1.3.4 представлены слоты класса «Музыкант». Слотами данного класса являются: страна происхождения, группа и имя.

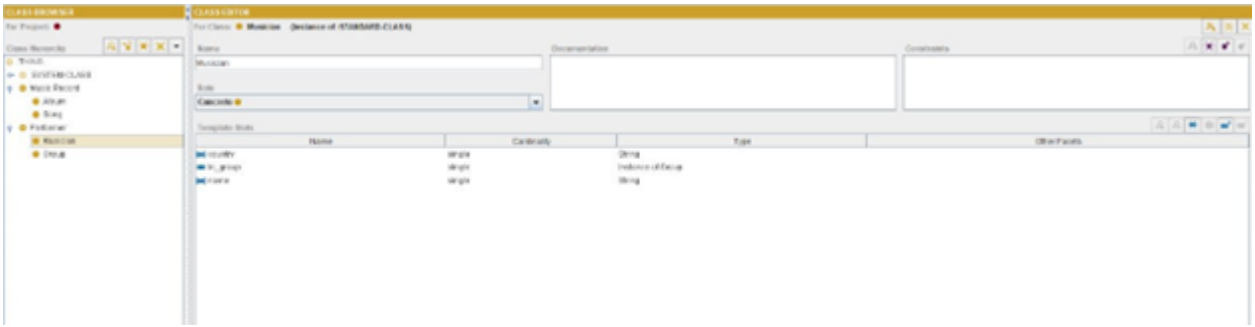

Рисунок 1.3.4 — Слоты класса «Музыкант»

На Рисунке 1.3.5 представлены слоты класса «Группа». Слотами данного класса являются страна и название.

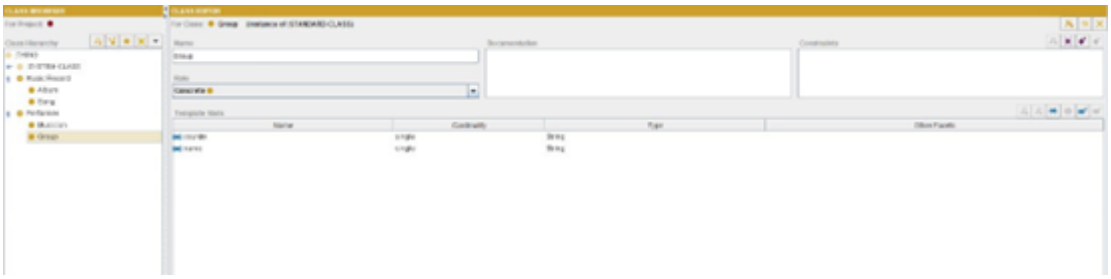

Рисунок 1.3.5 — Слоты класса «Группа»

После составления и описания классов созданы экземпляры каждого из классов. На Рисунке 1.3.6 представлены экземпляры класса «Группа» и значения полей в одном из них.

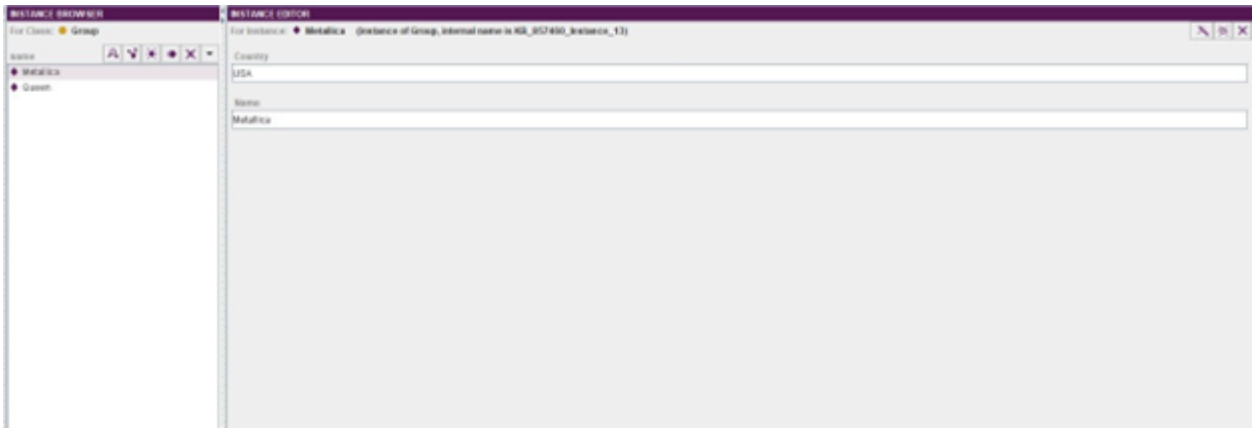

Рисунок 1.3.6 — Экземпляры класса «Группа»

На Рисунке 1.3.7 представлены экземпляры класса «Музыкант» и значения полей в одном из них.

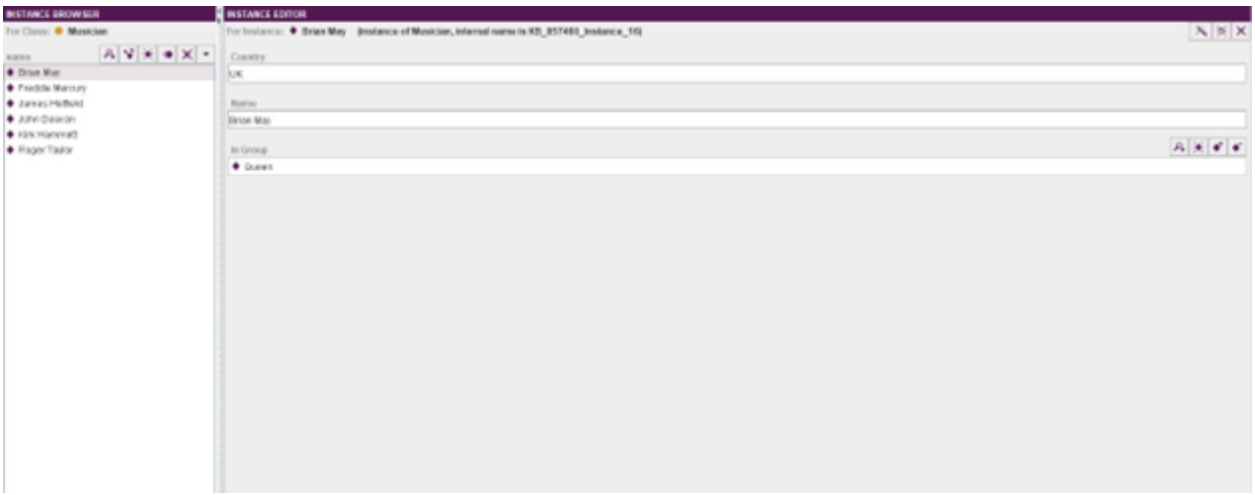

Рисунок 1.3.7 — Экземпляры класса «Музыкант»

На Рисунке 1.3.8 представлены экземпляры класса «Песня» и значения полей в одном из них.

Рисунок 1.3.8 — Экземпляры класса «Песня»

На Рисунке 1.3.9 представлены экземпляры класса «Альбом» и значения полей в одном из них.

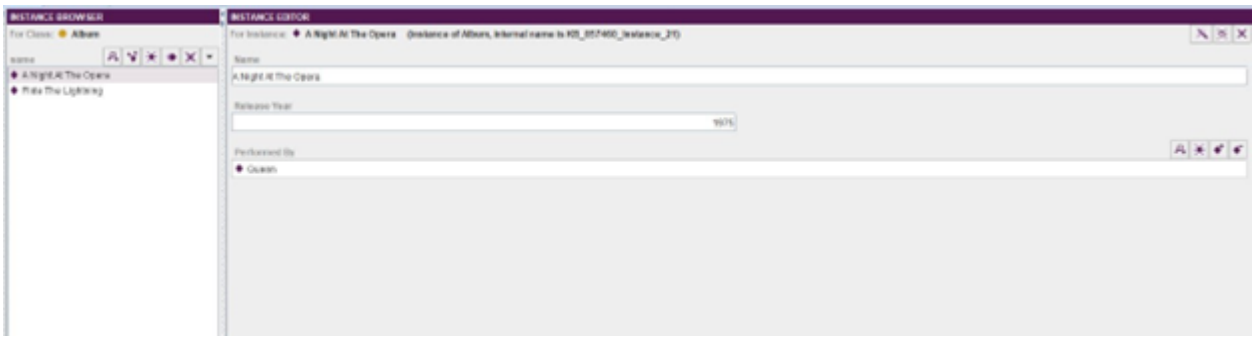

Рисунок 1.3.9 — Экземпляры класса «Альбом»

1.4 Выполнение запросов в Protege

Программа Protégé позволяет составлять запросы на получение объектов по определённым условиям, а также вытаскивать связанные объекты для уже полученных объектов. Проделан обычный запрос на получение экземпляров (Рисунок 1.4.1).

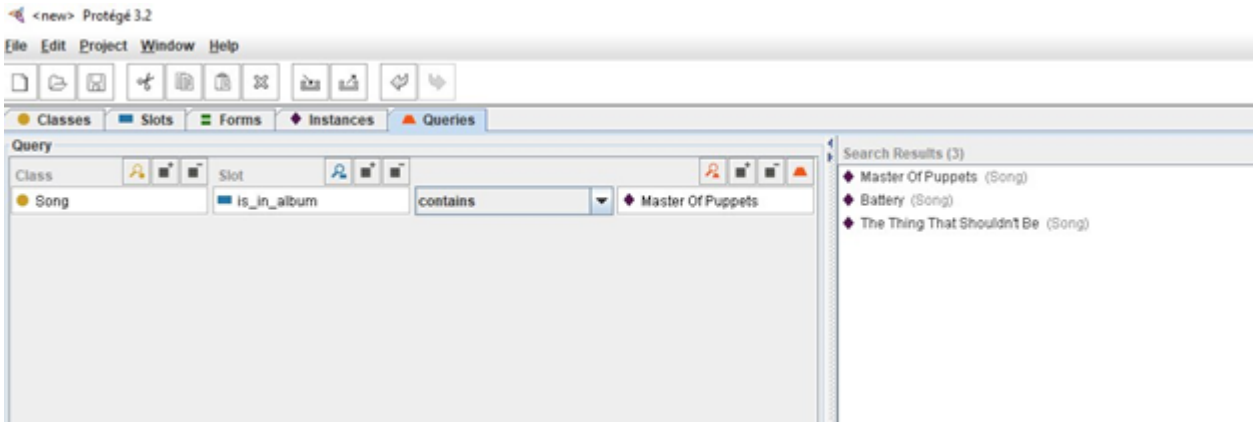

Рисунок 1.4.1 — Одинарный запрос на получение песен из альбома

На Рисунке 1.4.2 представлен цепной запрос на получение песен, написанных одной группой.

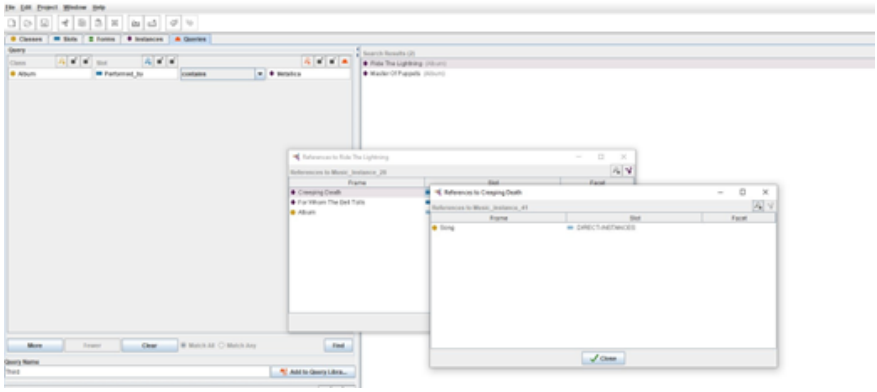

Рисунок 1.4.2 — Цепной запрос на получение песен, написанных одной группой

На Рисунке 1.4.3 представлен цепной запрос на получение песен, написанных одним музыкантом.

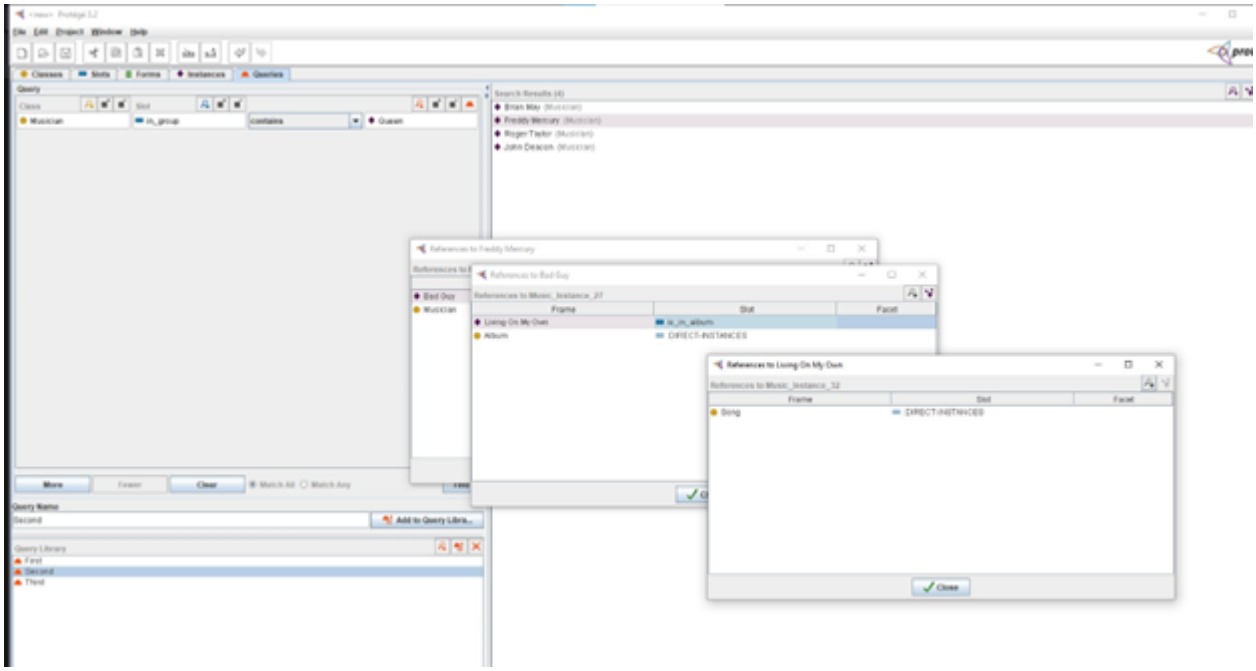

Рисунок 1.4.3 — Цепной запрос на получение песен, написанных одним музыкантом

1.5 Результаты выполнения программного кода

Для работы с онтологиями написана программа на языке Python, которая запускается в консоли и поддерживает выполнение запросов на получение экземпляров. Её код представлен в Листинге А.1. На Рисунке 1.5.1 представлен результат выполнения запроса музыкантов в группе, выполненный в программе.

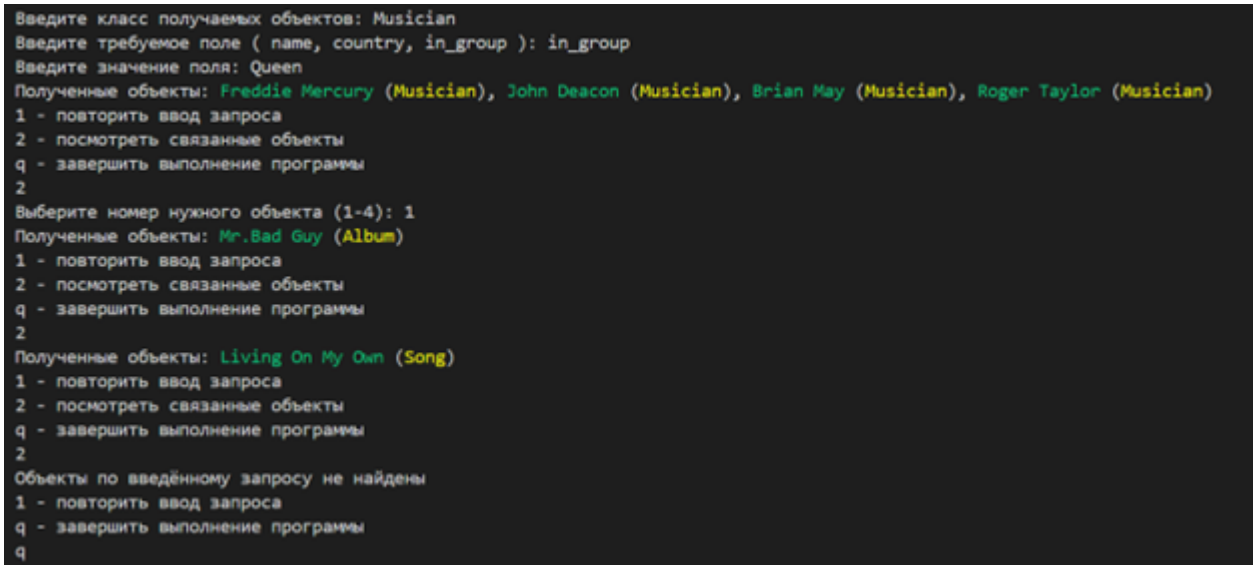

Рисунок 1.5.1 — Результат выполнения программы

На Рисунке 1.5.2 представлен результат получения песен, написанных группой, с помощью запроса, выполненного в программе.

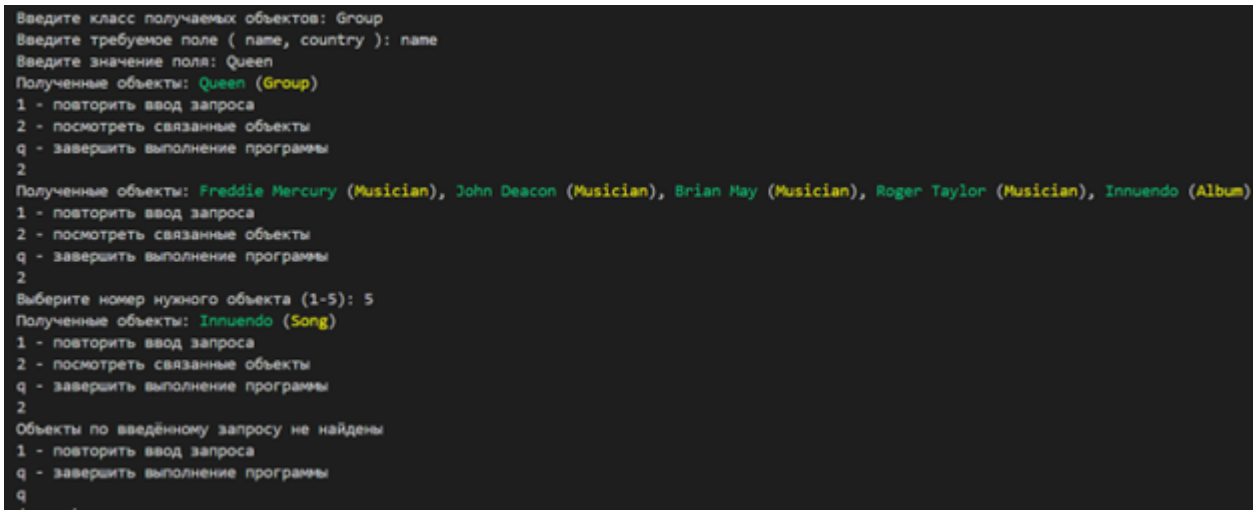

Рисунок 1.5.2 — Результат выполнения программы

1.6 Выводы по разделу

В ходе выполнения данной практической работы изучены теоретические основы системного анализа и использования онтологий в широком ряде задач,

получены навыки построения онтологий и работы с ними, включая создание классов для описания выбранной предметной области, создание слотов в классах и создание экземпляров. С помощью инструменты работы с онтологиями Protégé выполнены запросы на получение объектов по различным запросам.

В качестве закрепления полученных знаний написана программа на языке Python, способная работать с онтологией выбранной предметной области. В её функционал входит возможность писать запросы на получение экземпляров и связанных объектов.

2 МЕТОД ИМИТАЦИИ ОТЖИГА

Метод имитации отжига — это алгоритм оптимизации, основанный на принципах физического процесса отжига, в котором материал медленно охлаждается с целью достижения состояния минимальной энергии.

Алгоритм имитации отжига предложен в 1953 году Метрополисом (N.C. Metropolis). Данный алгоритм можно считать одним из немногих универсальных алгоритмов решения задач глобальной оптимизации.

Метод имитации отжига особенно актуален в современных условиях, когда многие задачи оптимизации стали слишком сложными для традиционных методов. С увеличением объёма данных и усложнением систем возникает необходимость в алгоритмах, способных находить решения в условиях высокой размерности и множества локальных минимумов. Имитация отжига предоставляет эффективный способ решения таких задач за счёт своей способности преодолевать локальные оптимумы и исследовать глобальные решения, что делает метод востребованным в разнообразных отраслях.

Метод имитации отжига находит широкое применение в таких сферах, как логистика и транспорт (например, для решения задач маршрутизации и оптимизации цепей поставок), телекоммуникации (для оптимизации сетевых ресурсов), экономика (для моделирования и оптимизации портфелей инвестиций), проектирование сложных инженерных систем (например, для оптимального размещения элементов на печатных платах), а также в биоинформатике и химии (для решения задач структурной биологии и моделирования молекул).

2.1 Описание алгоритма

Алгоритм имитации отжига состоит из следующих шагов:

1. Задание начальных параметров — выбор случайного начального решения и заданного значения температуры.

2. Генерация нового решения в окрестности текущего и оценка его качества.
3. Если новое решение лучше текущего, то алгоритм переходит к нему. Если нет, то переход всё равно может быть выполнен с некоторой вероятностью, зависящей от температуры и разности в качестве решений (2.1.1).

$$h(X,T) = \begin{cases} 1\text{ , если } f(X') - f(X) < 0 \\ e^{-\frac{\Delta E}{T}}\text{ , если } f(X') - f(X) \geq 0 \end{cases} \quad (2.1.1)$$

4. После этого шага температура снижается по заданному закону, и происходит переход к шагу 2.
5. Точкой останова алгоритма является достижение температурой определённого порога.

2.2 Постановка задачи

Цель работы: реализовать метод имитации отжига для решения задачи коммивояжёра и нахождения оптимального значения функции.

Поставлены следующие задачи:

- изучить метод имитации отжига;
- выбрать предметную область для задачи коммивояжёра и функцию для оптимизации;
- расписать ручной расчёт двух итераций в каждой из задач;
- разработать программный код решения задач методом имитации отжига.

Условие задачи коммивояжёра: дан полный граф, т.е. из каждой вершины можно пройти в любую другую вершину. В этом графе нужно найти полный путь минимальной длины, т.е. обойти каждую вершину в графе по одному разу.

Нахождение глобального минимума функции от многих переменных состоит в поиске точки в многомерном пространстве, где значение функции

будет минимальным. Сложность этой задачи состоит в том, что функция может содержать множество локальных минимумов, где производная функция равна нулю, но значение функции не является минимальным.

Выбранная предметная область для задачи коммивояжёра: в торговом центре расположено n магазинов. Человеку нужно пройтись по всем этим магазинам, при этом ему нужно затратить как можно меньше усилий на это, т.е. общий пройденный путь должен быть минимально возможным. Поэтому нужно определить минимальный путь, позволяющий обойти все магазины.

Выбранная функция для оптимизации: функция Растригина (2.2.1). Она примечательна тем, что имеет большое количество локальных минимумов. Глобальный минимум функции достигается в точке $(0; 0)$ и равен 0, при этом, в остальных локальных минимумах значение функции больше нуля. Функция рассматривается на области $x_i \in [-5.12, 5.12]$.

$$f(x, y) = 20 + x^2 - 10\cos(2\pi x) + y^2 - 10\cos(2\pi y) \qquad (2.2.1)$$

2.3 Задача коммивояжёра

2.3.1 Математическая модель

Для ручного расчёта число магазинов в задаче взято равным 6. Для каждого магазина случайным образом сгенерированы координаты в двумерном пространстве (диапазон координат от −10 до 10). Эти данные представлены заданы в Таблице 2.3.1.

Таблица 2.3.1 — Характеристики магазинов

Номер магазина	Координата по x	Координата по y
1	92	-97
2	-67	28
3	-14	-90
4	-67	72
5	19	75
6	77	-97

Важно, что у дома нулевые координаты.

Для расчёта расстояний между вершинами в графе использовалось Евклидово расстояние для точек в двумерном пространстве (2.3.1).

$$d = \sqrt{(x_2 - x_1)^2 - (y_2 - y_1)^2} \tag{2.3.1}$$

Ниже приведены расчёты длины каждого ребра в графе, т.е. рассчитаны длины путей между каждой парой вершин.

$$\sqrt{(92 + 67)^2 + (-97 - 28)^2} = 202.25$$
$$\sqrt{(92 + 14)^2 + (-97 + 90)^2} = 106.23$$
$$\sqrt{(92 + 67)^2 + (-97 - 72)^2} = 232.04$$
$$\sqrt{(92 - 19)^2 + (-97 - 75)^2} = 186.85$$
$$\sqrt{(92 - 77)^2 + (-97 + 97)^2} = 15.0$$
$$\sqrt{(-67 + 14)^2 + (28 + 90)^2} = 129.36$$
$$\sqrt{(-67 + 67)^2 + (28 - 72)^2} = 44.0$$
$$\sqrt{(-67 - 19)^2 + (28 - 75)^2} = 98.01$$
$$\sqrt{(-67 - 77)^2 + (28 + 97)^2} = 190.69$$
$$\sqrt{(-14 + 67)^2 + (-90 - 72)^2} = 170.45$$
$$\sqrt{(-14 - 19)^2 + (-90 - 75)^2} = 168.27$$
$$\sqrt{(-14 - 77)^2 + (-90 + 97)^2} = 91.27$$
$$\sqrt{(-67 - 19)^2 + (72 - 75)^2} = 86.05$$
$$\sqrt{(-67 - 77)^2 + (72 + 97)^2} = 222.03$$
$$\sqrt{(19 - 77)^2 + (75 + 97)^2} = 181.52$$

Рассчитанные длины рёбер сведены в Таблицу 2.3.2 с указанием вершин, составляющих ребро.

Таблица 2.3.2 — Длины рёбер в графе

Ребро	Длина ребра
0 → 1	202.25
0 → 2	106.23
0 → 3	232.04
0 → 4	186.85
0 → 5	15.00
1 → 2	129.36
1 → 3	44.00
1 → 4	98.01
1 → 5	190.69
2 → 3	170.45
2 → 4	168.27
2 → 5	91.27
3 → 4	86.05
3 → 5	222.03
4 → 5	181.52

2.3.2 Ручной расчёт

Сначала нужно составить путь случайным образом: $0 \to 4 \to 5 \to 3 \to 2 \to 6 \to 1 \to 0$. Длина полученного пути: $S_0 = 10.82 + 10.82 + 5.0 + 12.65 + 5.39 + 10.3 + 5.0 + 5.0 + 6.0 + 6.0 = 76.96(\text{м})$.

За изначальную температуру взята $T_0 = 100$.

Для первой итерации вместо четвёртого магазина поставлен пятый магазин, и после этого перестроен путь: $0 \to 5 \to 4 \to 3 \to 2 \to 6 \to 1 \to 0$. После этого проводится расчёт длины текущего пути: $S_1 = 5.0 + 7.81 + 15.81 + 10.3 + 8.06 + 8.06 + 10.82 + 10.82 + 5.0 + 5.0 = 86.68(\text{м})$. Длина текущего пути оказалась больше длины лучшего пути, поэтому проводится расчёт вероятности перехода к текущему решению.

$$H = e^{-\frac{\Delta l}{T_0}} = e^{-\frac{86.68-76.96}{100}} \approx 0.378$$

Вероятность, выданная псевдослучайным генератором чисел от 0 до 1, равна 0.363, что меньше 0.378. Поэтому текущее решение принимается как лучшее.

В конце первой итерации температура уменьшается в два раза по сравнению с изначальной: $T_1 = \frac{T_0}{2} = \frac{100}{2} = 50$.

Для второй итерации на место третьего магазина поставлен шестой, после чего перестроен путь: $0 \to 5 \to 1 \to 2 \to 6 \to 3 \to 4 \to 0$.

Длина получившегося пути равна $S_2 = 5.0 + 7.81 + 15.81 + 10.3 + 5.0 + 7.62 + 8.06 + 10.82 + 10.82 = 81.23(\text{м})$.

Длина получившегося пути меньше длины лучшего пути, поэтому текущее решение сразу принимается.

В конце второй итерации температура уменьшается в два раза по сравнению с температурой на текущей итерации: $T_2 = \frac{T_1}{2} = \frac{50}{2} = 25$.

2.4 Поиск глобального минимума

Выбранная функция: функция Растригина от двух переменных. Её формула представлена формулой 2.2.1.

На Рисунке 2.4.1 представлен график этой функции.

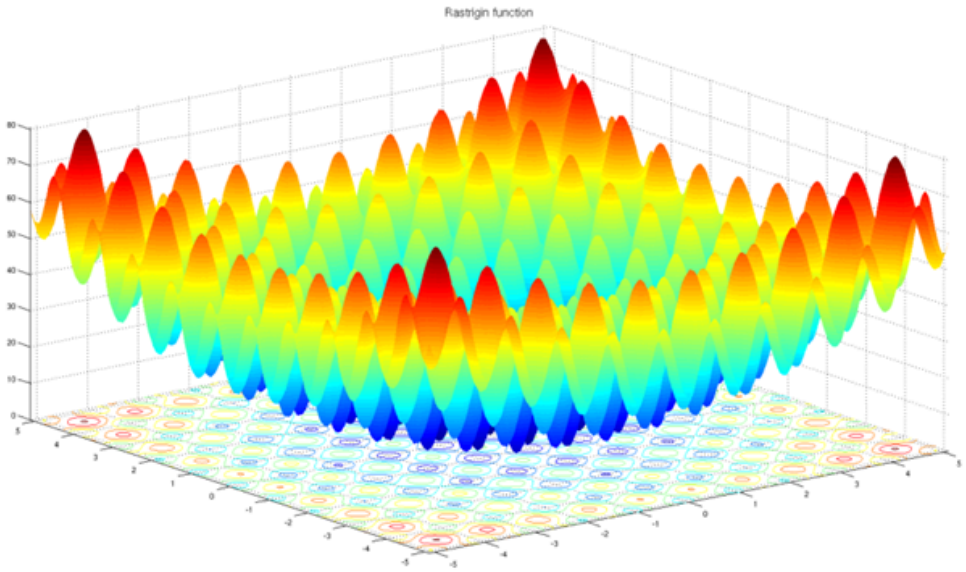

Рисунок 2.4.1 — График функции Растригина

Координаты начальной точки сгенерированы случайным образом и равны (2.75, 3.17). Значение функции в этой точке равно 32.53.

На первой итерации температура равна $T_0 = 100$.

Текущее решение на каждой итерации генерируется с использованием распределения Коши (2.4.1), где $D = 2$, т.к. задача рассматривается в двумерном пространстве.

$$g(x, x', T) = \frac{1}{\pi^D} \prod_{i=1}^{D} \frac{T}{|x' - x|^2 + T^2} \tag{2.4.1}$$

Текущее решение на первой итерации: (4.7, 5). Значение функции в этой точке равно 60.18.

Поскольку текущее решение оказалось хуже лучшего, то проводится расчёт вероятности перехода к этому решению.

$$H = e^{-\frac{\Delta f}{T_0}} = e^{-\frac{60.18-32.53}{100}} \approx 0.758$$

Вероятность, выданная псевдослучайным генератором чисел от 0 до 1, равна 0.923, что больше 0.758. Поэтому текущее решение отбрасывается. Лучшее решение после первой итерации: (2.75, 3.17), значение функции: 32.53.

После выполнения первой итерации температура изменена в соответствии с законом Коши.

$$T_1 = \frac{T_0}{k^{\frac{1}{D}}} = \frac{100}{1^{\frac{1}{2}}} = 100$$

При переходе на вторую итерацию текущим решением выбрано: (1.23, 1.35). Значение функции в этой точке равно 27.96.

Текущее решение оказалось оптимальнее лучшего, поэтому оно автоматически принимается. Лучшее решение после второй итерации: (1.23, 1.35), значение функции: 27.96.

После выполнения второй итерации температура изменена в соответствии с законом Коши.

$$T_2 = \frac{T_0}{k^{\frac{1}{D}}} = \frac{100}{2^{\frac{1}{2}}} \approx 70,711$$

2.5 Программная реализация

Для реализации расчётов метода имитации отжига написан программный код на языке Python.

В программной реализации задачи коммивояжёра зафиксированы следующие параметры:

- количество магазинов: 10;
- диапазон координат магазинов: $[-100; 100]$.

Код, реализующий решение поставленной задачи коммивояжёра методом имитации отжига, представлен в Листинге Б.1. Код, отвечающий за генерацию данных для задачи, представлен в Листинге Б.2, а сгенерированные данные представлены в Листинге Б.3.

На Рисунке 2.5.1 представлен результат выполнения программы, решающей задачу коммивояжёра — построенный граф после завершения алгоритма. Вертикальная и горизонтальная оси являются координатными осями.

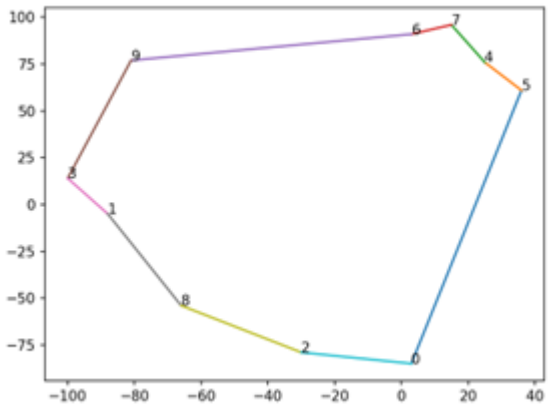

Рисунок 2.5.1 — Построенный минимальный путь в графе

На Рисунке 2.5.2 представлен график, отражающий изменение длины пути в графе в течение выполнения итераций алгоритма.

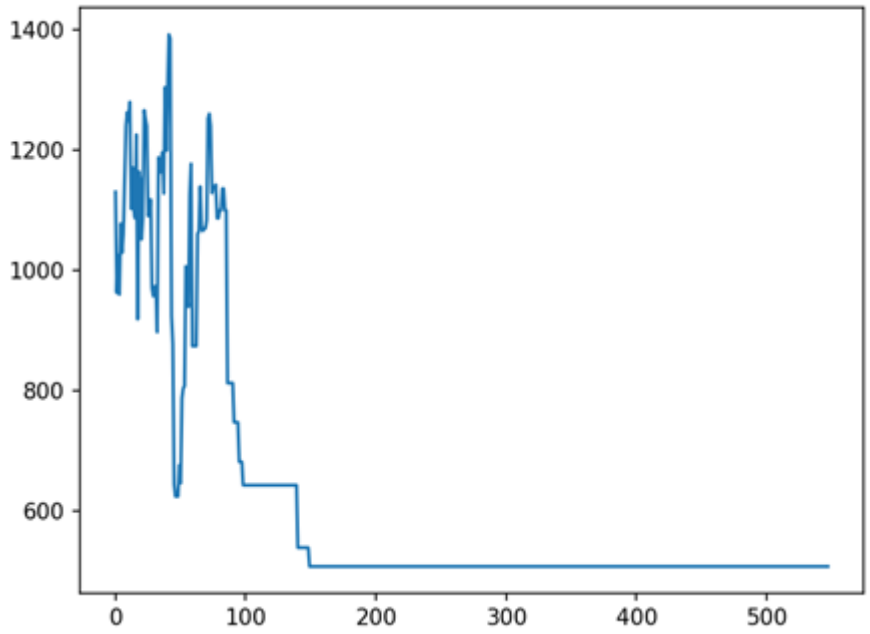

Рисунок 2.5.2 — График изменения длины пути

Код реализации имитации отжига методом Коши для нахождения оптимального значения функции представлен в Листинге В.1.

На Рисунке 2.5.3 представлен результат выполнения программы для нахождения оптимального значения функции — график зависимости оптимального решения от номера итерации.

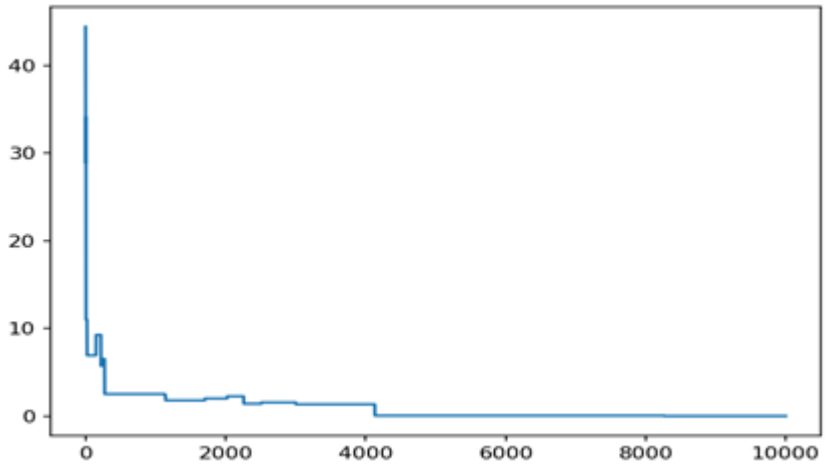

Рисунок 2.5.3 — График зависимости оптимального значения функции от номера итерации

2.6 Выводы по разделу

В ходе выполнения данной работы выполнены поставленные задачи – изучен метод имитации отжига, произведён его ручной расчёт для решения задачи коммивояжера и задачи поиска глобального минимума функции, а также разработаны программы на языке Python для решения поставленной задачи коммивояжёра – обхода всех магазинов, и для нахождения глобального минимума функции Растригина от двух переменных.

В заключение можно отметить, что метод имитации отжига является мощным инструментом для решения задач оптимизации, в которых стандартные методы недостаточно эффективны из-за наличия множества локальных минимумов. Благодаря своей способности к глобальному поиску и умению избегать застревания в локальных решениях, алгоритм находит широкое применение в различных областях, от логистики и сетевого планирования до биоинформатики и финансов.

3 АЛГОРИТМ РОЯ ЧАСТИЦ

В основу алгоритма оптимизации роем частиц положена социально-психологическая поведенческая модель толпы. Развитие алгоритма инспирировали такие задачи, как моделирование поведения птиц в стае и рыб в косяке. Целью было обнаружить базовые принципы, благодаря которым, например, птицы в стае ведут себя удивительно синхронно, меняя как по команде направления своего движения, так что стая движется как единое целое. К современному времени концепция алгоритма роя частиц развилась в высокоэффективный алгоритм оптимизации, часто составляющий конкуренцию лучшим модификациям генетического алгоритма.

В настоящее время роевой алгоритм применяются при решении задач численной и комбинаторной оптимизации, обучении искусственных нейронных сетей, построении нечетких контроллеров и т.д. в различных областях науки техники:

- управление энергетическими системами;
- решение NP-трудных комбинаторных проблем;
- задачи календарного планирования;
- оптимизация в мобильной связи;
- оптимизация процессов пакетной обработки;
- оптимизация многокритериальных задач;
- обработка изображений; распознавание образов;
- кластеризация данных;
- биоинформатика;
- проектирование сложных технических систем и т.д.

3.1 Описание алгоритма

Сначала происходит инициализация начальных параметров и роя — генерация точек в области поиска (количество точек задано и равно S), а также

свободных параметров алгоритма. Каждая точка имеет координаты и вектор скорости (3.1.1).

$$x_k = \left(x_1^k, x_2^k, \ldots, x_n^k\right); v_k = \left(v_1^k, v_2^k, \ldots, v_n^k\right), \quad (3.1.1)$$

где k — номер частицы;

n — размерность векторов в задаче.

Далее происходит поиск лучшего решения для каждой частицы, после которого обновляется лучшее решение для всего роя, если какой-то частицей найдено решение, которое лучше текущего. Затем выполняется коррекция скорости для каждой частицы по Формуле 3.1.2.

$$v_i(t+1) = v_i(t) + c_1 r_1(t)(y_i(t) - x_i(t)) + c_2 r_2(t)(\hat{y}(t) - x_i(t)), \quad (3.1.2)$$

где c_1, c_2 — положительные коэффициенты ускорения;

$r_1(t), r_2(t)$ — вектора размерности n, состоящие из случайных чисел из диапазона $(0; 1)$; при этом, $r_2(t) = 1 - r_1(t)$;

$y_i(t)$ — позиция i-й частицы, где достигалось лучшее решение;

$y(t)$ — координаты частицы с лучшим решением всего роя.

После этого выполняется коррекция позиции каждой частицы по Формуле 3.1.3.

$$x_i(t+1) = x_i(t) + v_i(t+1) \quad (3.1.3)$$

Точкой останова алгоритма является выполнение заданного числа итераций.

3.2 Постановка задачи

Цель работы: реализовать глобальный алгоритм роя частиц для нахождения оптимального значения функции.

Поставлены следующие задачи:

- изучить алгоритм роя частиц;
- выбрать тестовую функцию для оптимизации (нахождение глобального минимума);
- произвести ручной расчёт двух итераций алгоритма для трёх частиц;
- разработать программную реализацию алгоритма роя частиц для задачи минимизации функции.

Выбранная функция для оптимизации: функция Растригина (3.2.1). Она примечательна тем, что имеет большое количество локальных минимумов. Глобальный минимум функции достигается в точке $(0; 0)$ и равен 0, при этом, в остальных локальных минимумах значение функции больше нуля. Функция рассматривается на области $x_i \in [-5.12, 5.12]$.

$$f(x, y) = 20 + x^2 - 10\cos(2\pi x) + y^2 - 10\cos(2\pi y) \qquad (3.2.1)$$

3.3 Ручной расчёт алгоритма

Выбранная функция: функция Растригина от двух переменных. Её формула представлена Формулой 3.2.1. На Рисунке 3.3.1 представлен график этой функции.

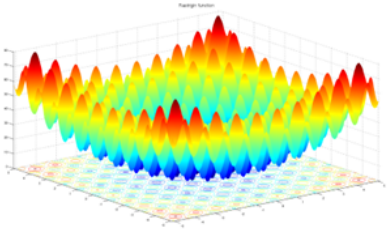

Рисунок 3.3.1 — График функции Растригина

Инициализированы свободные параметры алгоритма:

- $c_1 = c_2 = 2$;
- количество частиц: 3.

Далее созданы три частицы со следующими характеристиками:

$$x_1 = (0.2, 3.5); v_1 = (0, 0)$$
$$x_2 = (1.3, 0.98); v_2 = (0, 0)$$
$$x_3 = (4.87, -3.1); v_3 = (0, 0)$$

Значение целевой функции в первой точке равно 39.2; значение целевой функции у второй частицы равно 15.82; значение целевой функции у третьей частицы равно 38.39.

Лучшая позиция у каждой частицы пока что считается равной текущей позиции каждой частицы, а лучшая позиция всего роя — у второй частицы.

Затем выполняется коррекция скорости по Формуле 3.1.2. Для коррекции скорости первой частицы сгенерирован двумерный вектор из случайных чисел $r_1 = (0.234, 0.567)$.

$$v_{11}(1) = 0 + 2 * 0.234 * (0.2 - 0.2) + 2 * (1 - 0.234) * (1.3 - 0.2) = 1.685$$
$$v_{12}(1) = 0 + 2 * 0.567 * (3.5 - 3.5) + 2 * (1 - 0.567) * (0.98 - 3.5) = -2.182$$

Для коррекции скорости второй частицы сгенерирован двумерный вектор из случайных чисел $r_2 = (0.123, 0.987)$.

$$v_{21}(1) = 0 + 2 * 0.123 * (1.3 - 1.3) + 2 * (1 - 0.123) * (1.3 - 1.3) = 0$$
$$v_{22}(1) = 0 + 2 * 0.987 * (0.98 - 0.98) + 2 * (1 - 0.987) * (0.98 - 0.98) = 0$$

Для коррекции скорости третьей частицы сгенерирован двумерный вектор из случайных чисел $r_2 = (0.555, 0.002)$.

$$v_{31}(1) = 0 + 2 * 0.555 * (4.87 - 4.87) + 2 * (1 - 0.555)(1.3 - 4.87) = -3.177$$

$$v_{32}(1) = 0 + 2 * 0.002 * (-3.1 + 3.1) + 2 * (1 - 0.002) * (0.98 + 3.1) = 8.144$$

После коррекции скоростей выполняется коррекция координат каждой из частиц:

$$x_{11}(1) = 0.2 + 1.685 = 1.885$$
$$x_{12}(1) = 3.5 - 2.182 = 1.318$$
$$x_{21}(1) = 1.3 + 0 = 1.3$$
$$x_{22}(1) = 0.98 + 0 = 0.98$$
$$x_{31}(1) = 4.87 - 3.177 = 1.693$$
$$x_{32}(1) = -3.1 + 8.144 = 5.044$$

После этого происходит переход ко второй итерации. Заново рассчитаны значения функции у каждой частицы:

$$f_1(1) = 21.93$$
$$f_2(1) = 15.82$$
$$f_3(1) = 42.19$$

Значение целевой функции у первой частицы улучшилось, поэтому лучшая позиция теперь $(1.885, 1.318)$. Значение целевой функции у второй частицы не изменилось, поэтому её лучшая позиция осталась $(1.3, 0.98)$. Значение целевой функции у третьей частицы ухудшилось, поэтому лучшая позиция у третьей частицы остаётся как была изначально: $(4.87, -3.1)$. Лучшая позиция всего роя остаётся у второй частицы.

Затем выполняется коррекция скорости по Формуле 3.1.2. Для коррекции скорости первой частицы сгенерирован двумерный вектор из случайных чисел $r_1 = (0.124, 0.5)$.

$$v_{11}(2) = 1.685 + 2 * 0.124 * (1.885 - 1.885) +$$
$$+2 * (1 - 0.124) * (1.3 - 1.885) = 0.66$$

$$v_{12}(2) = 0 + 2 * 0.5 * (1.318 - 1.318) +$$
$$+2 * (1 - 0.5) * (0.98 - 1.318) = -0.33$$

Для коррекции скорости второй частицы сгенерирован двумерный вектор из случайных чисел $r_2 = (0.01, 0.8)$.

$$v_{21}(2) = 0 + 2 * 0.01 * (1.3 - 1.3) + 2 * (1 - 0.01) * (1.3 - 1.3) = 0$$
$$v_{22}(2) = 0 + 2 * 0.8 * (0.98 - 0.98) + 2 * (1 - 0.8) * (0.98 - 0.98) = 0$$

Для коррекции скорости третьей частицы сгенерирован двумерный вектор из случайных чисел $r_3 = (0.4, 0.8)$.

$$v_{31}(2) = 0 + 2 * 0.4 * (4.87 - 1.693) + 2 * (1 - 0.4) * (1.3 - 1.693) = 2.07$$
$$v_{32}(2) = 0 + 2 * 0.8 * (-3.1 - 5.044) + 2 * (1 - 0.8) * (0.98 - 5.044) = -14.656$$

После коррекции скоростей выполняется коррекция координат каждой из частиц:

$$x_{11}(1) = 1.885 + 0.66 = 2.545$$
$$x_{12}(1) = 1.318 - 0.33 = 0.988$$
$$x_{21}(1) = 1.3 + 0 = 1.3$$
$$x_{22}(1) = 0.98 + 0 = 0.98$$
$$x_{31}(1) = 1.693 + 2.07 = 3.763$$
$$x_{32}(1) = 5.044 - 14.656 = -9.612$$

Но координата x_{32} получилась вне области поиска, поэтому она принимается равной -5.12, т.е. позиция третьей частицы: $(3.763, -5.12)$. Заново рассчитаны значения функции у каждой частицы:

$$f_1(2) = 27.08$$
$$f_2(2) = 15.82$$

$$f_3(2) = 52.27$$

3.4 Программная реализация

Для реализации расчётов алгоритма роя частиц написан программный код на языке Python.

В программной реализации зафиксированы следующие параметры:

- количество частиц: 20;
- количество итераций: 30;
- c_1 и c_2: 2.

Код реализации роевого алгоритма для нахождения оптимального значения функции представлен в Листинге Г.1.

На Рисунке 3.4.1 представлен результат выполнения программы для нахождения оптимального значения функции — график зависимости оптимального решения от номера итерации.

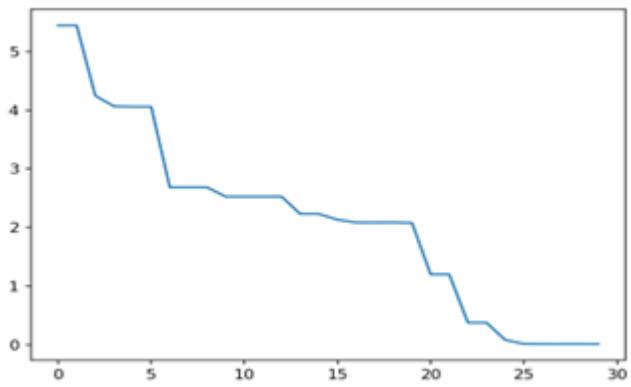

Рисунок 3.4.1 — График зависимости оптимального значения функции от номера итерации

3.5 Выводы по разделу

В ходе выполнения данной работы выполнены поставленные задачи — изучен алгоритм роя частиц, произведён его ручной расчёт для решения задачи поиска глобального минимума функции, а также разработаны программы на

языке Python для нахождения глобального минимума функции Растригина от двух переменных.

В заключение можно отметить, что роевой алгоритм является мощным инструментом для решения задач оптимизации, в которых стандартные методы недостаточно эффективны из-за наличия множества локальных минимумов. При этом, алгоритм является простым в реализации и имеет мало свободных параметров, из-за чего алгоритм не нуждается в длительной метаоптимизации.

4 МУРАВЬИНЫЙ АЛГОРИТМ

Муравьиный алгоритм — это эвристический метод оптимизации, разработанный итальянским ученым Марко Дориго в 1992 году. Работа алгоритма вдохновлена тем, как колония муравьев отправляется на поиски пищи. Каждый муравей оставляет на своем пути феромоны — химические вещества, привлекающие других муравьев. Чем больше муравьев проходит по определенному пути, тем сильнее концентрация феромонов на нем. В результате, большинство муравьев выбирает путь с наибольшей концентрацией феромонов, который, как правило, является кратчайшим.

Муравьиный алгоритм моделирует это поведение. Вместо реальных муравьев и феромонов, алгоритм использует «искусственных муравьев» и «искусственные феромоны».

Искусственные муравьи перемещаются по графу, представляющему пространство поиска, и оставляют феромоны на ребрах, которые они посещают. Вероятность выбора муравьем определенного ребра зависит от концентрации феромонов на нем и других факторов, таких как расстояние. Со временем, феромоны испаряются, что позволяет алгоритму «забывать» неудачные пути и сосредотачиваться на наиболее перспективных.

Муравьиный алгоритм широко используется для решения различных задач оптимизации, таких как:

- задача коммивояжера: поиск кратчайшего маршрута, проходящего через все заданные города;
- распределение ресурсов: оптимальное распределение ресурсов (например, рабочей силы, оборудования) для выполнения задач;
- планирование: составление расписаний, оптимизация маршрутов транспорта;
- сетевое проектирование: поиск оптимальных путей в сетях (например, компьютерных, транспортных).

4.1 Описание алгоритма

Сначала происходит инициализация начальных параметров и самой муравьиной колонии — для алгоритма муравьи создаются в количестве, равном количеству вершин в графе, и каждый из них начинает свой путь со своей вершины.

Важно отметить, что на каждую дугу графа «наносят» феромон — число, сгенерированное псевдослучайным образом в интервале от 0 до 1. Оно является одинаковым для всех дуг перед начальной итерацией.

Далее происходит построение пути для каждого муравья в колонии. Выбор муравьём новой вершины определяется с помощью вероятности, определяемой по Формуле 4.1.1.

$$p_{ij}^{k} = \begin{cases} \dfrac{\tau_{ij}^{\alpha}(t)\eta_{ij}^{\beta}(t)}{\sum_{u \in N_i^k} t_{iu}^{\alpha}(t)\eta_{iu}^{\beta}(t)}, \text{если } j \in N_i^k \\ 0, \text{если } j \notin N_i^k \end{cases}, \quad (4.1.1)$$

где i — номер текущей вершины муравья;

j — номер вершины, куда муравей может перейти;

τ_{ij} — количество феромона на дуге;

η_{ij} — априорная эффективность перехода по дуге из i в j;

N_i^k — множество доступных вершин для перемещения;

α, β — свободные параметры алгоритма (вес фермента и коэффициент эвристики.

После построения пути для каждого муравья высчитывается длина его пути.

Далее на каждой дуге происходит испарение феромона (4.1.2).

$$\tau_{ij}(t) = (1 - \rho)\tau_{ij}(t) \ , \quad (4.1.2)$$

где ρ — коэффициент испарения, $\rho \in [0, 1]$.

Для каждой дуги происходит изменение феромона в зависимости от того, насколько оптимальный путь получился у муравьёв (4.1.3).

$$\tau_{ij}(t+1) = t_{ij}(t) + \sum_{k=1}^{n_k} \Delta\tau_{ij}(t) \qquad (4.1.3)$$

где

$$\Delta\tau_{ij}^{k}(t) = \begin{cases} \frac{Q}{L^k}(t) \text{ , если дуга } (i,j) \text{ есть в пути } x^k(t) \\ 0 \quad \text{ , иначе} \end{cases}$$

— изменение количества феромона в зависимости от длины пройденного пути;

Q — положительная константа.

Точкой останова алгоритма является выполнение заданного числа итераций.

4.2 Постановка задачи

Цель работы: реализовать задачу коммивояжера муравьиным алгоритмом для нахождения приближённого оптимального маршрута.

Поставлены следующие задачи:

- изучить муравьиный алгоритм;
- выбрать предметную область для задачи коммивояжера;
- произвести ручной расчёт одной итерации алгоритма для двух муравьёв;
- разработать программную реализацию муравьиного алгоритма для задачи коммивояжера.

Условие задачи коммивояжёра: дан полный граф, т.е. из каждой вершины можно пройти в любую другую вершину. В этом графе нужно найти полный путь минимальной длины, т.е. обойти каждую вершину в графе по одному разу.

Выбранная предметная область для задачи коммивояжёра: в торговом центре расположено n магазинов. Человеку нужно пройтись по всем этим магазинам, при этом ему нужно затратить как можно меньше усилий на это, т.е. общий пройденный путь должен быть минимально возможным. Поэтому нужно определить минимальный путь, позволяющий обойти все магазины.

4.3 Ручной расчёт алгоритма

Для ручного расчёта число магазинов в задаче взято равным 6, число муравьёв — 2. Значения свободных параметров: $\alpha = 1, \beta = 5, \rho = 0.5$.

Для каждого магазина случайным образом сгенерированы координаты в двумерном пространстве (диапазон координат от −10 до 10). Эти данные представлены заданы в Таблице 4.3.3.

Таблица 4.3.3 — Характеристики магазинов

Номер магазина	Координата по x	Координата по y
1	92	-97
2	-67	28
3	-14	-90
4	-67	72
5	19	75
6	77	-97

Для расчёта расстояний между вершинами в графе использовалось Евклидово расстояние для точек в двумерном пространстве (4.3.1).

$$d = \sqrt{(x_2 - x_1)^2 - (y_2 - y_1)^2} \quad (4.3.1)$$

Ниже приведены расчёты длины каждого ребра в графе, т.е. рассчитаны длины путей между каждой парой вершин.

$$\sqrt{(92+67)^2+(-97-28)^2}=202.25$$
$$\sqrt{(92+14)^2+(-97+90)^2}=106.23$$
$$\sqrt{(92+67)^2+(-97-72)^2}=232.04$$
$$\sqrt{(92-19)^2+(-97-75)^2}=186.85$$
$$\sqrt{(92-77)^2+(-97+97)^2}=15.0$$
$$\sqrt{(-67+14)^2+(28+90)^2}=129.36$$
$$\sqrt{(-67+67)^2+(28-72)^2}=44.0$$
$$\sqrt{(-67-19)^2+(28-75)^2}=98.01$$
$$\sqrt{(-67-77)^2+(28+97)^2}=190.69$$
$$\sqrt{(-14+67)^2+(-90-72)^2}=170.45$$
$$\sqrt{(-14-19)^2+(-90-75)^2}=168.27$$
$$\sqrt{(-14-77)^2+(-90+97)^2}=91.27$$
$$\sqrt{(-67-19)^2+(72-75)^2}=86.05$$
$$\sqrt{(-67-77)^2+(72+97)^2}=222.03$$
$$\sqrt{(19-77)^2+(75+97)^2}=181.52$$

Рассчитанные длины рёбер сведены в Таблицу 4.3.4 с указанием вершин, составляющих ребро.

Таблица 4.3.4 — Длины рёбер в графе

Ребро	**Длина ребра**
0 → 1	202.25
0 → 2	106.23
0 → 3	232.04
0 → 4	186.85
0 → 5	15.00
1 → 2	129.36
1 → 3	44.00
1 → 4	98.01
1 → 5	190.69
2 → 3	170.45
2 → 4	168.27
2 → 5	91.27

Продолжение Таблицы 4.3.4

3 → 4	86.05
3 → 5	222.03
4 → 5	181.52

Также проинициализированы значения феромона на каждой дуге одним случайным значением: 0.876.

Для расчёта априорной эффективности из Формулы 4.1.1 использована Формула 4.3.2, чтобы иметь возможность проводить расчёты с точностью до пяти знаков после запятой.

$$\eta_{ij} = \frac{100}{d_{ij}} \tag{4.3.2}$$

4.3.1 Первая итерация

Для упрощения расчётов муравьи начинают свой путь с первой вершины.

Далее приведены расчёты вероятностей перехода первого муравья с 1-й вершины:

$$p_{12}^1 = \frac{0.876^1 * 0.494^2}{0.00403} = 0.005$$
$$p_{13}^1 = \frac{0.876^1 * 0.941^2}{0.00403} = 0.019$$
$$p_{14}^1 = \frac{0.876^1 * 0.431^2}{0.00403} = 0.004$$
$$p_{15}^1 = \frac{0.876^1 * 0.535^2}{0.00403} = 0.006$$
$$p_{16}^1 = \frac{0.876^1 * 6.667^2}{0.00403} = 0.965$$

Сгенерировано случайное число r = 0.783, которое определяет, на какую из вершин будет совершён переход. Для этого рассчитанные вероятности складываются в накопительную сумму.

$$P_2 = 0.005 < r = 0.783$$
$$P_3 = 0.025 < r = 0.783$$
$$P_4 = 0.029 < r = 0.783$$
$$P_5 = 0.035 < r = 0.783$$
$$P_6 = 1.000 \geq r = 0.783$$

Происходит переход на вершину 6. Далее приведены расчёты вероятностей перехода первого муравья с 6-й вершины:

$$p_{62}^1 = \frac{0.876^1 * 0.524^2}{0.00017} = 0.139$$
$$p_{63}^1 = \frac{0.876^1 * 1.096^2}{0.00017} = 0.606$$
$$p_{64}^1 = \frac{0.876^1 * 0.450^2}{0.00017} = 0.102$$
$$p_{65}^1 = \frac{0.876^1 * 0.551^2}{0.00017} = 0.153$$

Сгенерировано случайное число r = 0.044, которое определяет, на какую из вершин будет совершён переход. Для этого рассчитанные вероятности складываются в накопительную сумму.

$$P_2 = 0.139 \geq r = 0.044$$

Происходит переход на вершину 2. Далее приведены расчёты вероятностей перехода первого муравья с 2-й вершины:

$$p_{23}^1 = \frac{0.876^1 * 0.773^2}{0.00060} = 0.088$$
$$p_{24}^1 = \frac{0.876^1 * 2.273^2}{0.00060} = 0.759$$
$$p_{25}^1 = \frac{0.876^1 * 1.020^2}{0.00060} = 0.153$$

Сгенерировано случайное число r = 0.674, которое определяет, на какую из вершин будет совершён переход. Для этого рассчитанные вероятности складываются в накопительную сумму.

$$P_3 = 0.088 < r = 0.674$$
$$P_4 = 0.847 \geq r = 0.674$$

Происходит переход на вершину 4. Далее приведены расчёты вероятностей перехода первого муравья с 4-й вершины:

$$p_{43}^1 = \frac{0.876^1 * 0.587^2}{0.00015} = 0.203$$
$$p_{45}^1 = \frac{0.876^1 * 1.162^2}{0.00015} = 0.797$$

Сгенерировано случайное число r = 0.994, которое определяет, на какую из вершин будет совершён переход. Для этого рассчитанные вероятности складываются в накопительную сумму.

$$P_3 = 0.203 < r = 0.994$$
$$P_5 = 1.000 \geq r = 0.994$$

Происходит переход на вершину 5. Далее приведены расчёты вероятностей перехода первого муравья с 5-й вершины:

$$p_{53}^1 = \frac{0.876^1 * 0.594^2}{0.00003} = 1.000$$

Сгенерировано случайное число r = 0.545, которое определяет, на какую из вершин будет совершён переход. Для этого рассчитанные вероятности складываются в накопительную сумму.

$$P_3 = 1.000 \geq r = 0.545$$

Происходит переход на вершину 3. Затем муравей возвращается на начальную вершину, и его путь завершается. Таким образом, путь первого муравья выглядит следующим образом: $0 \to 5 \to 1 \to 3 \to 4 \to 2 \to 0$. Длина пути равна: $15.0 + 190.69 + 44.0 + 86.05 + 168.27 + 106.23 = 610.236$.

Далее приведены расчёты вероятностей перехода второго муравья с 1-й вершины:

$$p_{12}^2 = \frac{0.876^1 * 0.494^2}{0.00403} = 0.005$$
$$p_{13}^2 = \frac{0.876^1 * 0.941^2}{0.00403} = 0.019$$
$$p_{14}^2 = \frac{0.876^1 * 0.431^2}{0.00403} = 0.004$$
$$p_{15}^2 = \frac{0.876^1 * 0.535^2}{0.00403} = 0.006$$
$$p_{16}^2 = \frac{0.876^1 * 6.667^2}{0.00403} = 0.965$$

Сгенерировано случайное число r = 0.145, которое определяет, на какую из вершин будет совершён переход. Для этого рассчитанные вероятности складываются в накопительную сумму.

$$P_2 = 0.005 < r = 0.145$$
$$P_3 = 0.025 < r = 0.145$$
$$P_4 = 0.029 < r = 0.145$$
$$P_5 = 0.035 < r = 0.145$$
$$P_6 = 1.000 \geq r = 0.145$$

Происходит переход на вершину 6. Далее приведены расчёты вероятностей перехода второго муравья с 6-й вершины:

$$p_{62}^2 = \frac{0.876^1 * 0.524^2}{0.00017} = 0.139$$
$$p_{63}^2 = \frac{0.876^1 * 1.096^2}{0.00017} = 0.606$$
$$p_{64}^2 = \frac{0.876^1 * 0.450^2}{0.00017} = 0.102$$
$$p_{65}^2 = \frac{0.876^1 * 0.551^2}{0.00017} = 0.153$$

Сгенерировано случайное число r = 0.071, которое определяет, на какую из вершин будет совершён переход. Для этого рассчитанные вероятности складываются в накопительную сумму.

$$P_2 = 0.139 \geq r = 0.071$$

Происходит переход на вершину 2. Далее приведены расчёты вероятностей перехода второго муравья с 2-й вершины:

$$p_{23}^2 = \frac{0.876^1 * 0.773^2}{0.00060} = 0.088$$
$$p_{24}^2 = \frac{0.876^1 * 2.273^2}{0.00060} = 0.759$$
$$p_{25}^2 = \frac{0.876^1 * 1.020^2}{0.00060} = 0.153$$

Сгенерировано случайное число r = 0.919, которое определяет, на какую из вершин будет совершён переход. Для этого рассчитанные вероятности складываются в накопительную сумму.

$$P_3 = 0.088 < r = 0.919$$
$$P_4 = 0.847 < r = 0.919$$
$$P_5 = 1.000 \geq r = 0.919$$

Происходит переход на вершину 5. Далее приведены расчёты вероятностей перехода второго муравья с 5-й вершины:

$$p_{53}^2 = \frac{0.876^1 * 0.594^2}{0.00015} = 0.207$$
$$p_{54}^2 = \frac{0.876^1 * 1.162^2}{0.00015} = 0.793$$

Сгенерировано случайное число r = 0.548, которое определяет, на какую из вершин будет совершён переход. Для этого рассчитанные вероятности складываются в накопительную сумму.

$$P_3 = 0.207 < r = 0.548$$
$$P_4 = 1.000 \geq r = 0.548$$

Происходит переход на вершину 4. Далее приведены расчёты вероятностей перехода второго муравья с 4-й вершины:

$$p_{43}^2 = \frac{0.876^1 * 0.587^2}{0.00003} = 1.000$$

Сгенерировано случайное число r = 0.265, которое определяет, на какую из вершин будет совершён переход. Для этого рассчитанные вероятности складываются в накопительную сумму.

$$P_3 = 1.000 \geq r = 0.265$$

Происходит переход на вершину 3. Затем муравей возвращается на начальную вершину, и его путь завершается. Таким образом, путь второго муравья выглядит следующим образом: $0 \to 5 \to 1 \to 4 \to 3 \to 2 \to 0$. Длина пути равна: $15.0 + 190.69 + 98.01 + 86.05 + 170.45 + 106.23 = 666.423$.

Затем выполняется испарение феромона по Формуле 4.1.2:

$$\tau_{12} = (1 - 0.5) * 0.876 = 0.438$$
$$\tau_{13} = (1 - 0.5) * 0.876 = 0.438$$
$$\tau_{14} = (1 - 0.5) * 0.876 = 0.438$$
$$\tau_{15} = (1 - 0.5) * 0.876 = 0.438$$
$$\dots$$
$$\tau_{65} = (1 - 0.5) * 0.876 = 0.438$$

Изменение концентрации феромона происходит по Формуле 4.1.3, где значение Q принято за 100. Далее отображены только значения феромонов, которые после данной итерации изменились на ненулевую величину.

$$\tau_{16} = 0.438 + 0.314 = 0.752$$
$$\tau_{24} = 0.438 + 0.164 = 0.602$$
$$\tau_{25} = 0.438 + 0.150 = 0.588$$
$$\tau_{31} = 0.438 + 0.314 = 0.752$$
$$\tau_{43} = 0.438 + 0.150 = 0.588$$
$$\tau_{45} = 0.438 + 0.164 = 0.602$$
$$\tau_{53} = 0.438 + 0.164 = 0.602$$
$$\tau_{54} = 0.438 + 0.150 = 0.588$$
$$\tau_{62} = 0.438 + 0.314 = 0.752$$

4.3.2 Вторая итерация

Далее приведены расчёты вероятностей перехода первого муравья с 1-й вершины:

$$p_{12}^{1} = \frac{0.438^{1} * 0.494^{2}}{0.00341} = 0.003$$
$$p_{13}^{1} = \frac{0.438^{1} * 0.941^{2}}{0.00341} = 0.011$$

$$p_{14}^{1} = \frac{0.438^{1} * 0.431^{2}}{0.00341} = 0.002$$
$$p_{15}^{1} = \frac{0.438^{1} * 0.535^{2}}{0.00341} = 0.004$$
$$p_{16}^{1} = \frac{0.752^{1} * 6.667^{2}}{0.00341} = 0.979$$

Сгенерировано случайное число r = 0.954, которое определяет, на какую из вершин будет совершён переход. Для этого рассчитанные вероятности складываются в накопительную сумму.

$$P_2 = 0.003 < r = 0.954$$
$$P_3 = 0.015 < r = 0.954$$
$$P_4 = 0.017 < r = 0.954$$
$$P_5 = 0.021 < r = 0.954$$
$$P_6 = 1.000 \geq r = 0.954$$

Происходит переход на вершину 6. Далее приведены расчёты вероятностей перехода первого муравья с 6-й вершины:

$$p_{62}^{1} = \frac{0.752^{1} * 0.524^{2}}{0.00010} = 0.217$$
$$p_{63}^{1} = \frac{0.438^{1} * 1.096^{2}}{0.00010} = 0.551$$
$$p_{64}^{1} = \frac{0.438^{1} * 0.450^{2}}{0.00010} = 0.093$$
$$p_{65}^{1} = \frac{0.438^{1} * 0.551^{2}}{0.00010} = 0.139$$

Сгенерировано случайное число r = 0.098, которое определяет, на какую из вершин будет совершён переход. Для этого рассчитанные вероятности складываются в накопительную сумму.

$$P_2 = 0.217 \geq r = 0.098$$

Происходит переход на вершину 2. Далее приведены расчёты вероятностей перехода первого муравья с 2-й вершины:

$$p_{23}^1 = \frac{0.438^1 * 0.773^2}{0.00040} = 0.066$$
$$p_{24}^1 = \frac{0.602^1 * 2.273^2}{0.00040} = 0.781$$
$$p_{25}^1 = \frac{0.588^1 * 1.020^2}{0.00040} = 0.154$$

Сгенерировано случайное число r = 0.065, которое определяет, на какую из вершин будет совершён переход. Для этого рассчитанные вероятности складываются в накопительную сумму.

$$P_3 = 0.066 \geq r = 0.065$$

Происходит переход на вершину 3. Далее приведены расчёты вероятностей перехода первого муравья с 3-й вершины:

$$p_{34}^1 = \frac{0.438^1 * 0.587^2}{0.00003} = 0.494$$
$$p_{35}^1 = \frac{0.438^1 * 0.594^2}{0.00003} = 0.506$$

Сгенерировано случайное число r = 0.467, которое определяет, на какую из вершин будет совершён переход. Для этого рассчитанные вероятности складываются в накопительную сумму.

$$P_4 = 0.494 \geq r = 0.467$$

Происходит переход на вершину 4. Далее приведены расчёты вероятностей перехода первого муравья с 4-й вершины:

$$p_{45}^{1} = \frac{0.602^{1} * 1.162^{2}}{0.00008} = 1.000$$

Сгенерировано случайное число r = 0.958, которое определяет, на какую из вершин будет совершён переход. Для этого рассчитанные вероятности складываются в накопительную сумму.

$$P_5 = 1.000 \geq r = 0.958$$

Происходит переход на вершину 5. Затем муравей возвращается на начальную вершину, и его путь завершается. Таким образом, путь первого муравья выглядит следующим образом: $0 \to 5 \to 1 \to 2 \to 3 \to 4 \to 0$. Длина пути равна: $15.0 + 190.69 + 129.36 + 170.45 + 86.05 + 186.85 = 778.394$.

Далее приведены расчёты вероятностей перехода второго муравья с 1-й вершины:

$$p_{12}^{2} = \frac{0.438^{1} * 0.494^{2}}{0.00341} = 0.003$$
$$p_{13}^{2} = \frac{0.438^{1} * 0.941^{2}}{0.00341} = 0.011$$
$$p_{14}^{2} = \frac{0.438^{1} * 0.431^{2}}{0.00341} = 0.002$$
$$p_{15}^{2} = \frac{0.438^{1} * 0.535^{2}}{0.00341} = 0.004$$
$$p_{16}^{2} = \frac{0.752^{1} * 6.667^{2}}{0.00341} = 0.979$$

Сгенерировано случайное число r = 0.866, которое определяет, на какую из вершин будет совершён переход. Для этого рассчитанные вероятности складываются в накопительную сумму.

$$P_2 = 0.003 < r = 0.866$$
$$P_3 = 0.015 < r = 0.866$$
$$P_4 = 0.017 < r = 0.866$$
$$P_5 = 0.021 < r = 0.866$$
$$P_6 = 1.000 \geq r = 0.866$$

Происходит переход на вершину 6. Далее приведены расчёты вероятностей перехода второго муравья с 6-й вершины:

$$p_{62}^2 = \frac{0.752^1 * 0.524^2}{0.00010} = 0.217$$
$$p_{63}^2 = \frac{0.438^1 * 1.096^2}{0.00010} = 0.551$$
$$p_{64}^2 = \frac{0.438^1 * 0.450^2}{0.00010} = 0.093$$
$$p_{65}^2 = \frac{0.438^1 * 0.551^2}{0.00010} = 0.139$$

Сгенерировано случайное число r = 0.879, которое определяет, на какую из вершин будет совершён переход. Для этого рассчитанные вероятности складываются в накопительную сумму.

$$P_2 = 0.217 < r = 0.879$$
$$P_3 = 0.768 < r = 0.879$$
$$P_4 = 0.861 < r = 0.879$$
$$P_5 = 1.000 \geq r = 0.879$$

Происходит переход на вершину 5. Далее приведены расчёты вероятностей перехода второго муравья с 5-й вершины:

$$p_{52}^2 = \frac{0.438^1 * 1.020^2}{0.00015} = 0.312$$
$$p_{53}^2 = \frac{0.602^1 * 0.594^2}{0.00015} = 0.145$$
$$p_{54}^2 = \frac{0.588^1 * 1.162^2}{0.00015} = 0.543$$

Сгенерировано случайное число r = 0.343, которое определяет, на какую из вершин будет совершён переход. Для этого рассчитанные вероятности складываются в накопительную сумму.

$$P_2 = 0.312 < r = 0.343$$
$$P_3 = 0.457 \geq r = 0.343$$

Происходит переход на вершину 3. Далее приведены расчёты вероятностей перехода второго муравья с 3-й вершины:

$$p_{32}^2 = \frac{0.438^1 * 0.773^2}{0.00004} = 0.635$$
$$p_{34}^2 = \frac{0.438^1 * 0.587^2}{0.00004} = 0.365$$

Сгенерировано случайное число r = 0.329, которое определяет, на какую из вершин будет совершён переход. Для этого рассчитанные вероятности складываются в накопительную сумму.

$$P_2 = 0.635 \geq r = 0.329$$

Происходит переход на вершину 2. Далее приведены расчёты вероятностей перехода второго муравья с 2-й вершины:

$$p_{24}^2 = \frac{0.602^1 * 2.273^2}{0.00031} = 1.000$$

Сгенерировано случайное число r = 0.841, которое определяет, на какую из вершин будет совершён переход. Для этого рассчитанные вероятности складываются в накопительную сумму.

$$P_4 = 1.000 \geq r = 0.841$$

Происходит переход на вершину 4. Затем муравей возвращается на начальную вершину, и его путь завершается. Таким образом, путь второго муравья выглядит следующим образом: $0 \to 5 \to 4 \to 2 \to 1 \to 3 \to 0$. Длина пути равна: $15.0 + 181.52 + 168.27 + 129.36 + 44.0 + 232.04 = 770.178$.

Затем выполняется испарение феромона по Формуле 4.1.2 аналогично предыдущей итерации.

Изменение концентрации феромона происходит по Формуле 4.1.3, где значение Q принято за 100. Далее отображены только значения феромонов, которые после данной итерации изменились на ненулевую величину.

$$\tau_{16} = 0.376 + 0.258 = 0.634$$
$$\tau_{23} = 0.219 + 0.128 = 0.347$$
$$\tau_{24} = 0.301 + 0.130 = 0.431$$
$$\tau_{32} = 0.219 + 0.130 = 0.349$$
$$\tau_{34} = 0.219 + 0.128 = 0.347$$
$$\tau_{41} = 0.219 + 0.130 = 0.349$$
$$\tau_{45} = 0.301 + 0.128 = 0.429$$
$$\tau_{51} = 0.219 + 0.128 = 0.347$$
$$\tau_{53} = 0.301 + 0.130 = 0.431$$
$$\tau_{62} = 0.376 + 0.128 = 0.504$$
$$\tau_{65} = 0.219 + 0.130 = 0.349$$

4.4 Программная реализация

Для реализации расчётов алгоритма муравьиной колонии написан программный код на языке Python 3.12.

В программной реализации зафиксированы следующие параметры:

- количество вершин в графе: 10;
- количество муравьёв: 10;
- количество итераций: 40;
- $\alpha = 1$;
- $\beta = 5$;
- $\rho = 0.5$;
- $Q = 100$.

Код реализации алгоритма муравьиной колонии представлен в Листинге Д.1.

На Рисунке 4.4.1 представлен результат выполнения программы для нахождение минимального пути в графе — графики зависимости лучшего пути и среднего пути от номера итерации.

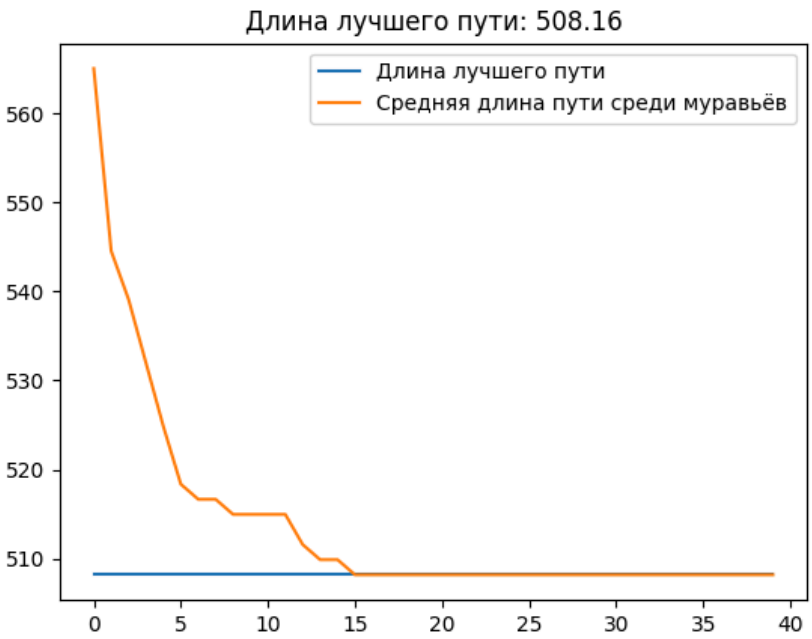

Рисунок 4.4.1 — График зависимости длины пути от номера итерации

На Рисунке 4.4.2 представлен лучший путь, построенный муравьями в результате выполнения алгоритма.

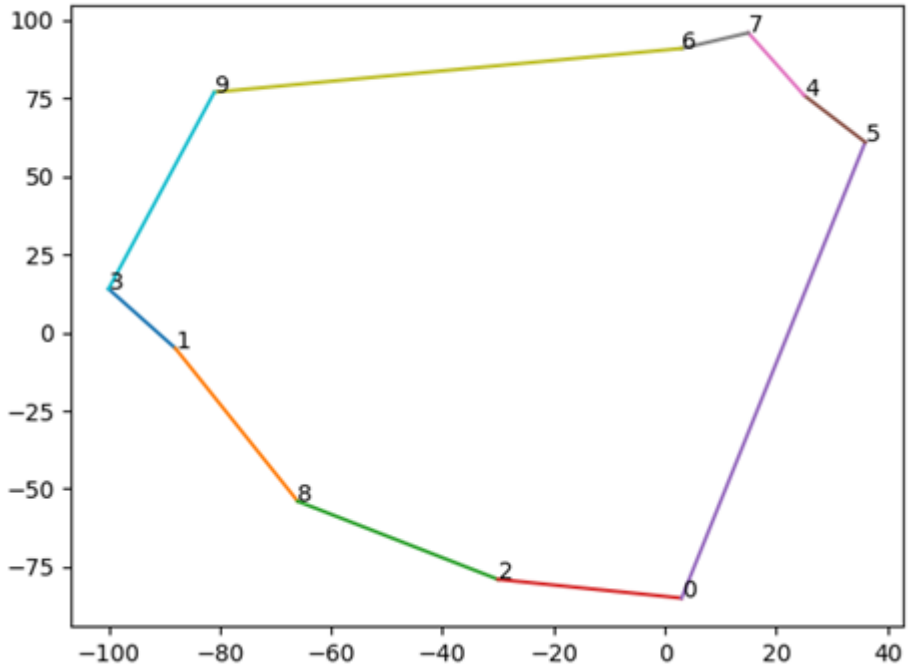

Рисунок 4.4.2 — Построенный путь

4.5 Выводы по разделу

В ходе выполнения данной работы выполнены поставленные задачи — изучен муравьиный алгоритм, произведён его ручной расчёт для решения задачи коммивояжёра, а также разработаны программы на языке Python для нахождения оптимального пути в графе.

В заключение можно отметить, что муравьиный алгоритм является мощным инструментом для решения задач оптимизации, в которых стандартные методы недостаточно эффективны из-за наличия множества решений, среди которых нужно найти только одно оптимальное. Алгоритм отлично справляется с нахождением оптимальных путей в графе, однако его сходимость является не очень высокой, а также результативность алгоритма сильно зависит от настройки свободных параметров.

5 АЛГОРИТМ ПЧЕЛИНОЙ КОЛОНИИ

Алгоритм пчелиной колонии — это эвристический метод оптимизации, разработанный Марко Дориго и Дино Д'Агостино в 2005 году. Этот алгоритм вдохновлен поведением медоносных пчел, которые демонстрируют удивительную способность находить наилучшие источники нектара для сбора меда.

Основной целью работы пчелиной колонии в природе является разведка пространства вокруг улья с целью поиска нектара с последующим его сбором. Для этого в составе колонии существуют различные типы пчел: пчелы-разведчики и рабочие пчелы-фуражиры (кроме них, в колонии существуют трутни и матка, не участвующие в процессе сбора нектара). Разведчики ведут исследование окружающего улей пространства и сообщают информацию о перспективных местах, в которых было обнаружено наибольшее количество нектара (для обмена информацией в улье существует специальный механизм, именуемый танцем пчелы).

Алгоритм пчелиной колонии моделирует это поведение. Вместо реальных пчел и танцев, алгоритм использует «искусственных пчел» и «искусственные танцы». Искусственные пчелы перемещаются по пространству поиска, представленному в виде графа или сетки, и оценивают качество каждой позиции. Затем они возвращаются в «улей» и передают информацию о найденных позициях другим пчелам. Вероятность выбора пчелой определенной позиции зависит от ее качества и количества информации, полученной от других пчел. Со временем, пчелы концентрируют свои усилия на наиболее перспективных позициях.

Алгоритм пчелиной колонии широко используется для решения различных задач оптимизации, таких как:

- задача календарного планирования;
- задача коммивояжёра;
- транспортная задача.

5.1 Описание алгоритма

Сначала происходит инициализация начальных параметров и пчёл – генерация точек в области поиска (количество точек задано и равно S), а также свободных параметров алгоритма. Каждая точка имеет координаты (5.1.1).

$$X_j = \left(x_{1j}, x_{2j}, \ldots, x_{nj}\right) \ , \tag{5.1.1}$$

где $j \in [1; S]$ — номер частицы;

n — размерность векторов в задаче.

Формирование подобластей происходит на основе Евклидова расстояния между пчёлами (5.1.2).

$$d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2} \tag{5.1.2}$$

Сначала выбирается точка с наименьшим значением функции – она становится центром новой подобласти. Вокруг неё собираются все пчёлы, расстояние до которых от центральной пчелы меньше заданного числа ε. После проверки всех пчёл убираются те пчёлы, которые вошли в подобласть, и данные действия повторяются для оставшихся пчёл.

После формирования подобластей начинается поиск оптимального значения в каждой из них. В каждой области выбирается точка с наилучшим значение функции, вокруг неё в квадрате со стороной 2Δ генерируются случайным образом $S - 1$ пчёл, а затем среди сгенерированных пчёл и центральной пчелы выбирается та, которая имеет наименьшее значение функции. Теперь эта точка становится центром новой области, и процесс повторяется до тех пор, пока не наилучшая точка не останется статичной в течение заданного числа итераций.

Такой поиск проводится в каждой из полученных подобластей, и точкой останова алгоритма является окончание поиска в последней области.

Точкой останова алгоритма является выполнение заданного числа итераций.

5.2 Постановка задачи

Цель работы: реализовать глобальный алгоритм пчелиной колонии для нахождения оптимального значения функции.

Поставлены следующие задачи:

- изучить алгоритм пчелиной колонии;
- выбрать тестовую функцию для оптимизации (нахождение глобального минимума);
- произвести ручной расчёт одной итерации алгоритма;
- разработать программную реализацию алгоритма пчелиной колонии для задачи минимизации функции.

Выбранная функция для оптимизации: функция Растригина (5.2.1). Она примечательна тем, что имеет большое количество локальных минимумов. Глобальный минимум функции достигается в точке $(0; 0)$ и равен 0, при этом, в остальных локальных минимумах значение функции больше нуля. Функция рассматривается на области $x_i \in [-5.12, 5.12]$.

$$f(x, y) = 20 + x^2 - 10\cos(2\pi x) + y^2 - 10\cos(2\pi y) \qquad (5.2.1)$$

5.3 Ручной расчёт алгоритма

Выбранная функция: функция Растригина от двух переменных. Её формула представлена Формулой 5.2.1. На Рисунке 5.3.1 представлен график этой функции.

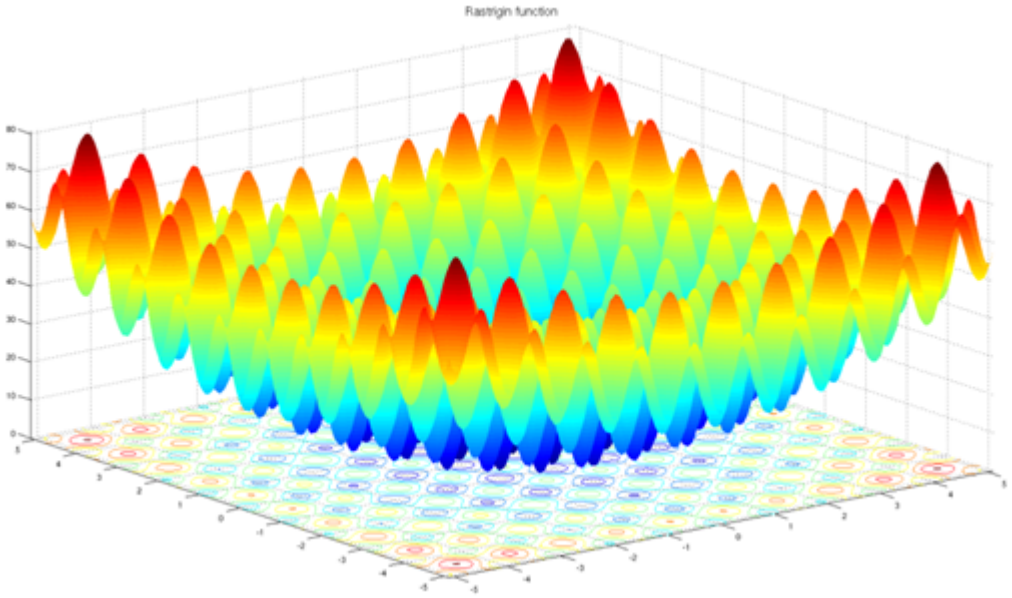

Рисунок 5.3.1 — График функции Растригина

Инициализированы свободные параметры алгоритма:

- $\varepsilon = 2$;
- $\Delta = 1$;
- количество пчёл-разведчиков (S): 8.

Создано 8 пчёл со следующими координатами:

$$X_1 = (-4.078, -5.091); f(X_0) = 45.310$$
$$X_2 = (4.723, -2.923); f(X_1) = 43.654$$
$$X_3 = (3.593, 3.706); f(X_2) = 57.682$$
$$X_4 = (-0.960, 3.632); f(X_3) = 31.198$$
$$X_5 = (3.945, -2.769); f(X_4) = 32.614$$
$$X_6 = (1.065, 4.941); f(X_5) = 27.056$$
$$X_7 = (4.918, 4.094); f(X_6) = 43.920$$
$$X_8 = (-1.068, 1.670); f(X_7) = 19.633$$

Среди оставшихся точек лучшее значение имеется у пчелы X_8: значение функции у неё равно 19.633. Далее рассчитывается Евклидово расстояние между точкой X_8 и оставшимися точками по Формуле 5.2.1.

$$d_{81} = \sqrt{(-1.068 + 4.078)^2 + (1.670 + 5.091)^2} = 7.401 \geq 2$$

$$d_{82} = \sqrt{(-1.068 - 4.723)^2 + (1.670 + 2.923)^2} = 7.392 \geq 2$$
$$d_{83} = \sqrt{(-1.068 - 3.593)^2 + (1.670 - 3.706)^2} = 5.087 \geq 2$$
$$d_{84} = \sqrt{(-1.068 + 0.960)^2 + (1.670 - 3.632)^2} = 1.964 < 2$$
$$d_{85} = \sqrt{(-1.068 - 3.945)^2 + (1.670 + 2.769)^2} = 6.696 \geq 2$$
$$d_{86} = \sqrt{(-1.068 - 1.065)^2 + (1.670 - 4.941)^2} = 3.904 \geq 2$$
$$d_{87} = \sqrt{(-1.068 - 4.918)^2 + (1.670 - 4.094)^2} = 6.458 \geq 2$$

Следовательно, в область точки X_8 вошла точка X_4.

Среди оставшихся точек лучшее значение имеется у пчелы X_6: значение функции у неё равно 27.056.

Далее рассчитывается Евклидово расстояние между точкой X_6 и оставшимися точками по Формуле 5.2.1.

$$d_{61} = \sqrt{(1.065 + 4.078)^2 + (4.941 + 5.091)^2} = 11.273 \geq 2$$
$$d_{62} = \sqrt{(1.065 - 4.723)^2 + (4.941 + 2.923)^2} = 8.673 \geq 2$$
$$d_{63} = \sqrt{(1.065 - 3.593)^2 + (4.941 - 3.706)^2} = 2.813 \geq 2$$
$$d_{65} = \sqrt{(1.065 - 3.945)^2 + (4.941 + 2.769)^2} = 8.230 \geq 2$$
$$d_{67} = \sqrt{(1.065 - 4.918)^2 + (4.941 - 4.094)^2} = 3.945 \geq 2$$

Следовательно, точка X_6 образует область сама с собой.

Среди оставшихся точек лучшее значение имеется у пчелы X_5: значение функции у неё равно 32.614.

Далее рассчитывается Евклидово расстояние между точкой X_5 и оставшимися точками по Формуле 5.2.1.

$$d_{51} = \sqrt{(3.945 + 4.078)^2 + (-2.769 + 5.091)^2} = 8.352 \geq 2$$
$$d_{52} = \sqrt{(3.945 - 4.723)^2 + (-2.769 + 2.923)^2} = 0.794 < 2$$
$$d_{53} = \sqrt{(3.945 - 3.593)^2 + (-2.769 - 3.706)^2} = 6.485 \geq 2$$

$$d_{57} = \sqrt{(3.945 - 4.918)^2 + (-2.769 - 4.094)^2} = 6.932 \geq 2$$

Следовательно, в область точки X_5 вошла точка X_2.

Среди оставшихся точек лучшее значение имеется у пчелы X_7: значение функции у неё равно 43.920.

Далее рассчитывается Евклидово расстояние между точкой X_7 и оставшимися точками по Формуле 5.2.1.

$$d_{71} = \sqrt{(4.918 + 4.078)^2 + (4.094 + 5.091)^2} = 12.856 \geq 2$$
$$d_{73} = \sqrt{(4.918 - 3.593)^2 + (4.094 - 3.706)^2} = 1.380 < 2$$

Следовательно, в область точки X_7 вошла точка X_3.

Среди оставшихся точек лучшее значение имеется у пчелы X_1: значение функции у неё равно 45.310.

Поскольку точек больше не осталось, то точка X_1 образует область сама с собой.

Рассмотрим поиск в первой подобласти. Лучшая точка: $(-1.068, 1.670)$ со значением функции 19.633. Новые сгенерированные точки имеют следующие координаты (точка X_8 является текущим центром области):

$$X_1 = (-0.822, 0.917); f(X_0) = 8.453$$
$$X_2 = (-1.212, 2.105); f(X_1) = 15.630$$
$$X_3 = (-0.196, 1.449); f(X_2) = 28.284$$
$$X_4 = (-0.845, 0.943); f(X_3) = 6.618$$
$$X_5 = (-0.775, 0.699); f(X_4) = 22.633$$
$$X_6 = (-0.826, 2.319); f(X_5) = 25.707$$
$$X_7 = (-1.464, 2.197); f(X_6) = 33.421$$
$$X_8 = (-1.068, 1.670); f(X_7) = 19.633$$

Минимальное значение среди достигнуто точкой X_4 (значение функции равно 6.618). Следовательно, эта точка становится центром области, и происходит переход к новой итерации.

5.4 Программная реализация

Для реализации расчётов алгоритма пчелиной колонии написан программный код на языке Python.

В программной реализации зафиксированы следующие параметры:

- количество пчёл: 100;
- количество итераций: 30;
- $\varepsilon = 0.8$;
- $\Delta = 1$.

Код реализации алгоритма пчелиной колонии для нахождения оптимального значения функции представлен в Листинге Е.1.

На Рисунке 5.4.1 представлен результат выполнения программы для нахождения оптимального значения функции – консольный вывод результатов поиска в нескольких областях.

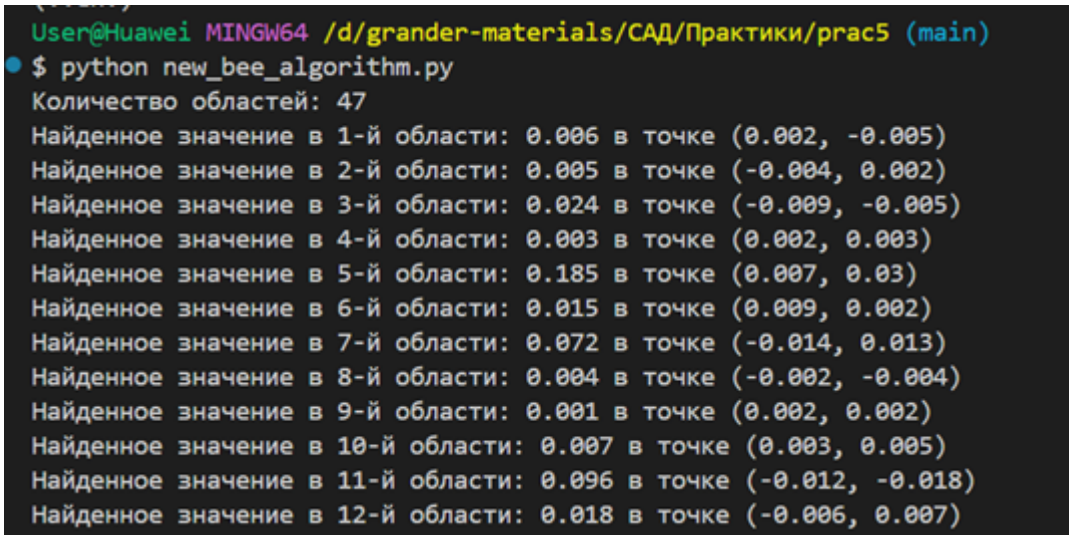

Рисунок 5.4.1 — Результаты поиска в первых 12 областях

5.5 Выводы по разделу

В ходе выполнения данной работы выполнены поставленные задачи — изучен алгоритм пчелиной колонии, произведён его ручной расчёт для решения задачи поиска глобального минимума функции, а также разработана программа на языке Python для нахождения глобального минимума функции Растригина от двух переменных.

В заключение можно отметить, что алгоритм пчелиной колонии является мощным инструментом для решения задач оптимизации (в том числе, задач нахождения глобального минимума функции), в которых стандартные методы недостаточно эффективны из-за наличия множества локальных минимумов. Алгоритм имеет высокую сходимость, однако его результативность сильно зависит от настройки большого количества свободных параметров.

6 ЭЛЕКТРОМАГНИТНЫЙ АЛГОРИТМ

Электромагнитный алгоритм предложен Бирбилом (I. Birbil) и Фангом (S.C. Fang) в 2003 году [2]. Этот алгоритм вдохновлен фундаментальными принципами электромагнетизма, а именно взаимодействием между электрическими зарядами и магнитными полями.

В алгоритме пространство поиска представляется как заполненное частицами, каждая из которых обладает определенным электрическим зарядом. Эти частицы взаимодействуют друг с другом посредством электромагнитных сил. Сила притяжения или отталкивания между двумя частицами зависит от их зарядов и расстояния между ними.

Каждого из агентов популяции в электромагнитном алгоритме интерпретируют как заряженную частицу, заряд которой пропорционален значению фитнесс-функции в той точке области поиска, в которой на данной итерации находится агент. Текущий заряд частиц популяции определяет суммарную силу, действующую на данную частицу со стороны других частиц, а также направление и величину её перемещений на текущей итерации. В соответствии с законами электростатики эта сила вычисляется путём векторного суммирования сил притяжения и отталкивания со стороны всех частиц популяции [2].

Электромагнитный алгоритм широко используется для решения различных задач оптимизации, таких как:

- задача коммивояжера;
- распределение ресурсов;
- планирование;
- сетевое проектирование.

6.1 Описание алгоритма

Сначала происходит инициализация начальных параметров и зарядов – генерация точек в области поиска (количество точек задано и равно S), а также свободных параметров алгоритма. Каждая точка имеет координаты 6.1.1.

$$X_j = \left(x_{1j}, x_{2j}, \ldots, x_{nj}\right) \tag{6.1.1}$$

где $j \in [1; S]$ — номер частицы;

n — размерность векторов в задаче.

Затем начинается локальный поиск – для каждого заряда выполняется линейный стохастический поиск 6.1.2 в целях сбора локальной информации об окружении частиц [2, с.22-23].

$$x'_{ij} = x_{ij} + u^{\text{sign}} * U(0;1) * \alpha(x^+ - x^-) \tag{6.1.2}$$

где u^{sign} — случайное целое число, равное -1 или 1;

$U(0;1)$ — случайное число от 0 до 1 из равномерного распределения;

$\alpha \in (0;1)$ — свободный параметр алгоритма;

x^+, x^- — границы рассматриваемой области поиска.

При этом, изменённая координата принимается только в том случае, если её изменение дало улучшение значения целевой функции, иначе происходит переход к следующей итерации локального поиска.

Как правило, свободный параметр α задаётся близким к нулю (порядка $\sim 10^{-2} : 10^{-4}$), поэтому локальный поиск служит только для незначительного улучшения положения точек в пространстве.

После осуществления локального поиска вычисляется лучшая точка, т.е. точка, где достигается минимальное среди всех значение целевой функции.

Затем для каждой частицы вычисляется её заряд по Формуле 6.1.3.

$$q_i = \exp\left(-n * \frac{(\varphi_i - \varphi_{\text{best}})}{\sum_{j,j\neq i}(\varphi_j - \varphi_{\text{best}})}\right) \quad (6.1.3)$$

где φ_i — значение целевой функции в текущей точке;

φ_{best} — значение целевой функции в лучшей точке.

На основе посчитанных зарядов происходит вычисление силы отталкивания и притяжения между частицами (6.1.4). Частица i отталкивается от частицы j, если значение функции у частицы i лучше, чем значение функции у частицы j, и наоборот, притягивается, если значение функции хуже.

$$F_i = \sum_{j=1,j\neq i}^{S} F_{i,j} = \sum_{j=1,j\neq i}^{S} \begin{cases} (X_j - X_i)\frac{q_i q_j}{\|X_j - X_i\|^2}, \text{если } \varphi_j < \varphi_i \\ (X_i - X_j)\frac{q_i q_j}{\|X_j - X_i\|^2}, \text{если } \varphi_i < \varphi_j \end{cases} \quad (6.1.4)$$

Таким образом, лучшая частица на данной итерации притянет к себе все остальные. Далее выполняется перемещение частиц, высчитываемое на основе электромагнитных сил (6.1.5).

$$X_i(t+1) = X_i(t) + U(0;1)\frac{F_i}{\|F_i\|}V_i \quad (6.1.5)$$

где $U(0;1)$ — случайное число от 0 до 1 из равномерного распределения;

$\frac{F_i}{\|F_i\|}$ — нормированный вектор силы;

V_i — вектор скорости, компоненты которого рассчитываются по Формуле 6.1.6.

$$v_{ij} = \begin{cases} (x^+ - x_{ij}), F_{ij} > 0 \\ (x_{ij} - x^-), F_{ij} \leq 0 \end{cases}, j \in [1:S], j \neq i \quad (6.1.6)$$

где x^+, x^- — границы области поиска.

Важно отметить, что лучшая частица должна остаться на своём месте, поэтому она в данной итерации не передвигается, а только притягивает к себе другие частицы.

После осуществления перемещения частиц происходит переход к следующей итерации. Точкой останова алгоритма является достижение максимального числа итераций.

6.2 Постановка задачи

Цель работы: реализовать электромагнитный алгоритм для нахождения оптимального значения функции.

Поставлены следующие задачи:

- изучить электромагнитный алгоритм;
- выбрать тестовую функцию для оптимизации (нахождение глобального минимума);
- произвести ручной расчёт одной итерации алгоритма;
- разработать программную реализацию электромагнитного алгоритма для задачи минимизации функции.

Выбранная функция для оптимизации: функция Растригина (6.2.1). Она примечательна тем, что имеет большое количество локальных минимумов. Глобальный минимум функции достигается в точке $(0; 0)$ и равен 0, при этом, в остальных локальных минимумах значение функции больше нуля. Функция рассматривается на области $x_i \in [-5.12, 5.12]$.

$$f(x, y) = 20 + x^2 - 10\cos(2\pi x) + y^2 - 10\cos(2\pi y) \qquad (6.2.1)$$

6.3 Ручной расчёт алгоритма

Выбранная функция: функция Растригина от двух переменных. Её формула представлена Формулой 6.2.1. На Рисунке 6.3.1 представлен график этой

функции.

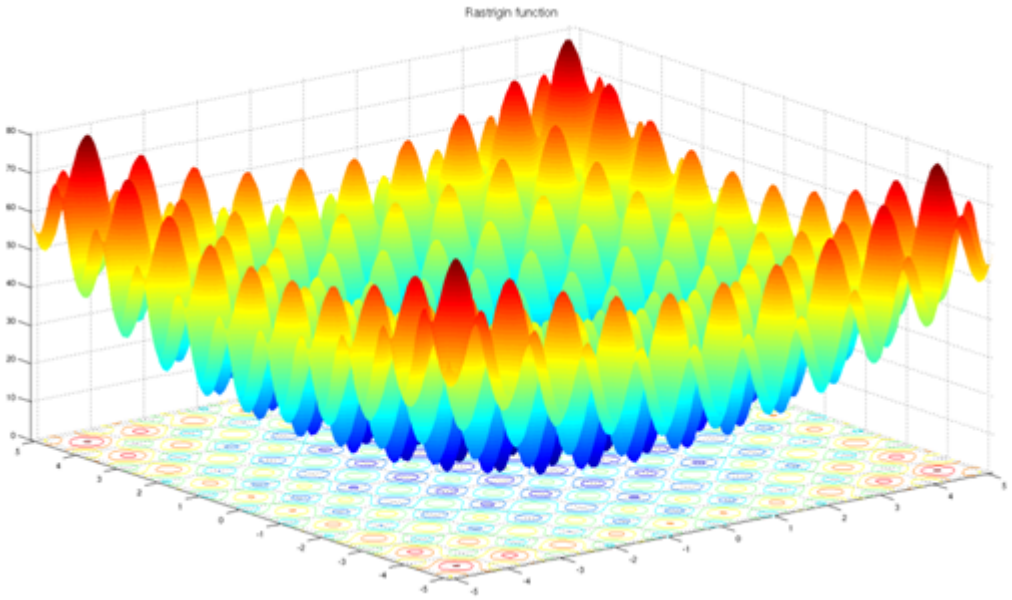

Рисунок 6.3.1 — График функции Растригина

Количество частиц, используемых для ручного расчёта: 4.

Создано 4 частицы со следующими координатами:

$$X_1 = (-1.911, 2.375); f(X_0) = 27.905$$
$$X_2 = (3.325, 4.537); f(X_1) = 65.893$$
$$X_3 = (3.839, -2.436); f(X_2) = 44.592$$
$$X_4 = (0.141, -1.939); f(X_3) = 8.157$$

После выполнения локального поиска (6.1.2) точки имеют следующие координаты:

$$X_1 = (-1.911, 2.368); f(X_0) = 27.550$$
$$X_2 = (3.291, 4.537); f(X_1) = 63.706$$
$$X_3 = (3.873, -2.436); f(X_2) = 43.152$$
$$X_4 = (0.141, -1.961); f(X_3) = 7.813$$

При этом, значение φ_{best} равно 7.813 в точке $(0.141, -1.961)$. Далее приведены расчёты значения заряда для каждой из частиц по Формуле 6.1.3:

$$q_1 = \exp\left(-2 * \frac{27.550 - 7.813}{110.971}\right) = 0.701$$

$$q_2 = \exp\left(-2 * \frac{63.706 - 7.813}{110.971}\right) = 0.365$$

$$q_3 = \exp\left(-2 * \frac{43.152 - 7.813}{110.971}\right) = 0.529$$

При этом, заряд у лучшей частицы будет равен единице, поскольку аргумент в экспоненте будет равен нулю. Затем вычисляются значение силы для каждой частицы по Формуле 6.1.4:

$$F_1 = 0.701 * 0.365 * \frac{(-1.911, 2.368) - (3.291, 4.537)}{31.766} +$$
$$+0.701 * 0.529 * \frac{(-1.911, 2.368) - (3.873, -2.436)}{56.543} +$$
$$+0.701 * 1.000 * \frac{(0.141, -1.961) - (-1.911, 2.368)}{22.953} = (-0.017, -0.118)$$

$$F_2 = 0.365 * 0.701 * \frac{(-1.911, 2.368) - (3.291, 4.537)}{31.766} +$$
$$+0.365 * 0.529 * \frac{(3.873, -2.436) - (3.291, 4.537)}{48.965} +$$
$$+0.365 * 1.000 * \frac{(0.141, -1.961) - (3.291, 4.537)}{52.152} = (-0.062, -0.09)$$

$$F_3 = 0.529 * 0.701 * \frac{(-1.911, 2.368) - (3.873, -2.436)}{56.543} +$$
$$+0.529 * 0.365 * \frac{(3.873, -2.436) - (3.291, 4.537)}{48.965} +$$
$$+0.529 * 1.000 * \frac{(0.141, -1.961) - (3.873, -2.436)}{14.160} = (-0.175, 0.022)$$

$$F_4 = 1.000 * 0.701 * \frac{(0.141, -1.961) - (-1.911, 2.368)}{22.953} +$$
$$+1.000 * 0.365 * \frac{(0.141, -1.961) - (3.291, 4.537)}{52.152} +$$
$$+1.000 * 0.529 * \frac{(0.141, -1.961) - (3.873, -2.436)}{14.160} = (-0.099, -0.16)$$

И наконец, позиции частиц изменяются по Формулам 6.1.5 - 6.1.6. Частица X_4 не передвигается, т.к. она имеет лучшее значение.

Рассчитаем смещение для 1-й частицы. Сгенерировано случайное число $U(0;1) = 0.845$.

Нормированный вектор силы у 1-й частицы: $F_1 = (-0.144, -0.99)$.

Рассчитаны компоненты вектора скорости для 1-й частицы:

$$v_{11} = -1.911 + 5.12 = 3.209$$
$$v_{12} = 2.368 + 5.12 = 7.488$$

Тогда частица сместится на следующую позицию:

$$X_1 = (-1.911, 2.368) + 0.845 * (-0.144, -0.99) * (3.209, 7.488)$$
$$= (-2.301, -3.891)$$
$$F(X_1) = 35.864$$

Рассчитаем смещение для 2-й частицы. Сгенерировано случайное число $U(0;1) = 0.403$ Нормированный вектор силы у 2-й частицы: $F_2 = (-0.563, -0.826)$ Рассчитаны компоненты вектора скорости для 2-й частицы:

$$v_{21} = 3.291 + 5.12 = 8.411$$
$$v_{22} = 4.537 + 5.12 = 9.657$$

Тогда частица сместится на следующую позицию:

$$X_2 = (3.291, 4.537) + 0.403 * (-0.563, -0.826) * (8.411, 9.657)$$
$$= (1.38, 1.317)$$
$$F(X_2) = 35.025$$

Рассчитаем смещение для 3-й частицы. Сгенерировано случайное число $U(0;1) = 0.767$ Нормированный вектор силы у 3-й частицы: $F_3 = (-0.992, 0.123)$ Рассчитаны компоненты вектора скорости для 3-й частицы:

$$v_{31} = 3.873 + 5.12 = 8.993$$
$$v_{32} = 5.12 + 2.436 = 7.556$$

Тогда частица сместится на следующую позицию:

$$X_3 = (3.873, -2.436) + 0.767 * (-0.992, 0.123) * (8.993, 7.556)$$
$$= (-2.973, -1.722)$$
$$F(X_3) = 23.689$$

6.4 Программная реализация

Для реализации расчётов электромагнитного алгоритма написан программный код на языке Python.

В программной реализации зафиксированы следующие параметры:

- количество частиц: 20;
- количество итераций локального поиска: 10;
- количество общих итераций: 100;
- $\alpha = 0.005$.

Код реализации электромагнитного алгоритма для нахождения оптимального значения функции представлен в Листинге Ж.1.

На Рисунке 6.4.1 представлен результат выполнения программы для нахождения оптимального значения функции – график зависимости минимального оптимального решения от номера итерации.

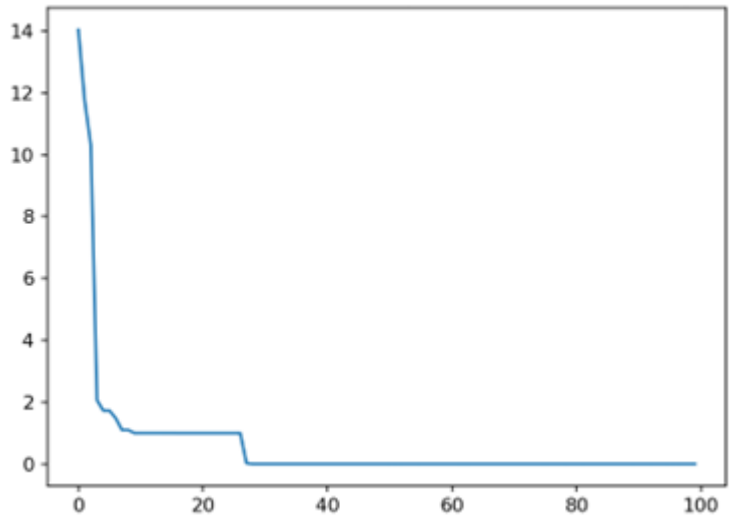

Рисунок 6.4.1 — Результат выполнения программы

6.5 Выводы по разделу

В ходе выполнения данной работы выполнены поставленные задачи – изучен электромагнитный алгоритм, произведён его ручной расчёт для решения задачи поиска глобального минимума функции, а также разработаны программы на языке Python для нахождения глобального минимума функции Растригина от двух переменных.

В заключение можно отметить, что электромагнитный алгоритм является мощным инструментом для решения задач оптимизации (например, для нахождения глобального минимума функции от нескольких переменных), в которых стандартные методы недостаточно эффективны из-за наличия множества локальных минимумов. Алгоритм имеет меньшую зависимость от свободных параметров, чем пчелиный алгоритм, но большую, чем роевой алгоритм. В то же время, алгоритм показал быструю сходимость, однако имеет тенденцию сходится к локальным минимумам.

ЗАКЛЮЧЕНИЕ

В ходе выполнения работы изучены основы системного анализа данных и применения онтологий, а также изучены и реализованы различные методы оптимизации, вдохновлённые природой.

Разработана онтология музыкальной индустрии, включающая классы, слоты и экземпляры. Онтология включает в себя 4 класса, позволяющие описать простые отношения внутри выбранной области. Для описания онтологии использован инструмент Protégé, также он используется для написания запросов на получение объектов по различным атрибутам. Также написана программа на языке Python для описания онтологии и написания запросов.

Для решения задачи коммивояжёра и задачи оптимизации функций от многих переменных реализовано несколько алгоритмов, позволяющих быстро найти оптимальное решение в поставленной задаче.

Метод имитации отжига является простым алгоритмом, который существенно полагается на случайную генерацию, однако за счёт механизма отжига, когда при высокой температуре может быть принято неоптимальное решение, алгоритм показывает хорошие результаты при решении как задачи коммивояжёра, так и задачи нахождения глобального минимума функции от нескольких переменных.

Роевой алгоритм предназначен для оптимизации функций, но находит более оптимальное решение, чем метод имитации отжига. При этом, он является простым в реализации и не требует сложной метаоптимизации.

Муравьиный алгоритм предназначен для решения задачи коммивояжёра, и имеет хорошую сходимость к оптимальному решению. Однако значительным недостатком алгоритма является зависимость от большого количества свободных аргументов, поскольку при неверной их комбинации алгоритм показывает неудовлетворительные результаты.

Алгоритм пчелиной колонии предназначен для нахождения глобального минимума функций. Алгоритм имеет хорошую сходимость и лучше сходится к

оптимальному решению по сравнению с роевым алгоритмом, однако недостатком является зависимость от регулирования свободных параметров.

Электромагнитный алгоритм предназначен для оптимизации функций. Свободные параметры меньше влияют на данный алгоритм по сравнению с алгоритмом пчелиной колонии, при этом алгоритм показывает лучшие показатели сходимости к глобальному минимуму.

Все описанные алгоритмы реализованы на языке Python для решения задачи коммивояжёра в произвольном графе и нахождения глобального минимума функции Растригина, которая примечательна наличием большого количества локальных минимумов.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Карпенко, А. П. Современные алгоритмы поисковой оптимизации. Алгоритмы, вдохновленные природой: учебное пособие / А. П. Карпенко — 3-е изд. — Москва: Издательство МГТУ им. Н. Э. Баумана, 2021. — 446 с.

2. Пряжников, В. Алгоритм имитации отжига [Электронный ресурс]. URL: https://pryazhnikov.com/notes/simulated-annealing/ (Дата обращения: 12.11.2024).

3. Сорокин, А. Б. Введение в роевой интеллект: теория, расчеты и приложения [Электронный ресурс]: Учебно-методическое пособие / А. Б. Сорокин — М.: Московский технологический университет (МИРЭА), 2019.

4. Ant colony optimization algorithms [Электронный ресурс]: Википедия. URL: https://en.wikipedia.org/wiki/Ant_colony_optimization_algorithms (Дата обращения: 01.11.2024).

5. Bees algorithm [Электронный ресурс]: Википедия. URL: https://en.wikipedia.org/wiki/Bees_algorithm (Дата обращения: 01.11.2024).

6. Rastrigin function [Электронный ресурс]: Википедия. URL: https://en.wikipedia.org/wiki/Rastrigin_function (Дата обращения: 01.11.2024).

7. Wang, Q., Zeng, J., Song, W. A New Electromagnetism-like Algorithm with Chaos Optimization 2010. С. 535–538.

ПРИЛОЖЕНИЯ

Приложение А — Код реализации онтологии на языке Python.

Приложение Б — Код реализации метода имитации отжига в задаче коммивояжёра на языке Python.

Приложение В — Код реализации метода имитации отжига в задаче оптимизации функции на языке Python.

Приложение Г — Реализация роевого алгоритма в задаче оптимизации на языке Python.

Приложение Д — Реализация муравьиного алгоритма на языке Python.

Приложение Е — Реализация алгоритма пчелиной колонии на языке Python.

Приложение Ж — Реализация электромагнитного алгоритма на языке Python.

Приложение А

Код реализации онтологии на языке Python

Листинг А.1 — Код файла main.py

```
from typing import Any
import random
import re


class OntologyObject:
    shown_field = 'name'

    def __str__(self) -> str:
        return getattr(self, self.shown_field)


class Performer(OntologyObject):
    instances = []

    def __init__(self, name: str, country: str) -> None:
        self.name = name
        self.country = country
        Performer.instances.append(self)


class MusicRecord(OntologyObject):
    instances = []

    def __init__(self, name: str):
        self.name = name
        MusicRecord.instances.append(self)


class Group(Performer):
    instances = []

    def __init__(self, name: str, country: str):
        super().__init__(name, country)
        Group.instances.append(self)


class Musician(Performer):
    instances = []

    def __init__(self, name: str, country: str, in_group: Group):
        super().__init__(name, country)
        self.in_group = in_group
        Musician.instances.append(self)


class Album(MusicRecord):
    instances = []

    def __init__(self, name: str, release_year: int, performed_by: Performer):
        super().__init__(name)
        self.release_year = release_year
        self.performed_by = performed_by
        Album.instances.append(self)


class Song(MusicRecord):
    instances = []

    def __init__(self, name: str, in_album: Album):
        super().__init__(name)
        self.in_album = in_album
        Song.instances.append(self)
```

Продолжение Листинга А.1

```
def find_related_objects_by_value(cls: OntologyObject.__class__,
                                  lookup_field: str, value: Any):
    instances = cls.instances
    result = []
    for instance in instances:
        if isinstance(getattr(instance, lookup_field), OntologyObject):
            if not isinstance(value, OntologyObject):
                value = find_object_by_name(
                    getattr(instance, lookup_field).__class__, value)
                if value is None:
                    return None

        if getattr(instance, lookup_field) == value:
            result.append((instance, type(instance).__name__))
    return result


def get_class(class_name: str):
    classes = {
        'musician': Musician,
        'group': Group,
        'album': Album,
        'song': Song
    }
    class_name = class_name.lower().strip()
    if class_name in classes:
        return classes[class_name]
    else:
        return None


def find_object_by_name(cls: OntologyObject.__class__, name: str):
    instances = cls.instances
    for instance in instances:
        if instance.name == name:
            return instance

    return None


def get_random_class_instance(cls: OntologyObject.__class__):
    instances = cls.instances
    return random.choice(instances)


def get_related_class(obj: OntologyObject):
    classes = [Album, Song, Musician, Group]
    class_types = set()
    for cls in classes:
        cls_instances = cls.instances
        for instance in cls_instances:
            fields = [
                key for key in instance.__dict__.keys()
                if not re.match(r"__\w*__", key)
            ]
            for field in fields:
                field_value = getattr(instance, field)
                if field_value == obj:
                    class_types.add((cls, field))
    return list(class_types)


def main():
    groups = [Group('Queen', 'Great Britain'), Group('Metallica', 'USA')]
    musicians = [
        Musician('Freddie Mercury', 'Zanzibar', groups[0]),
        Musician('John Deacon', 'UK', groups[0]),
        Musician('Brian May', 'UK', groups[0]),
        Musician('Roger Taylor', 'UK', groups[0]),
        Musician('James Hetfield', 'USA', groups[1]),
        Musician('Kirk Hammett', 'USA', groups[1]),
```

Продолжение Листинга А.1

```
        Musician('Lars Ulrich', 'USA', groups[1]),
        Musician('Robert Trujilio', 'USA', groups[1])
    ]
    albums = [
        Album('Mr.Bad Guy', 1985, musicians[0]),
        Album('A Night At The Opera', 1975, groups[1]),
        Album('Innuendo', 1991, groups[0]),
        Album('Ride The Lightning', 1984, groups[1]),
        Album('Master Of Puppets', 1986, groups[1])
    ]
    songs = [
        Song('Living On My Own', albums[0]),
        Song('Bohemiarn Rhapsody', albums[1]),
        Song('Love Of My Life', albums[1]),
        Song('Innuendo', albums[2]),
        Song('Ride The Lightning', albums[3]),
        Song('For Wthom The Bell Tolls', albums[3]),
        Song('Master Of Puppets', albums[4]),
        Song('Battery', albums[4])
    ]

    while True:
        while True:
            class_name = input('Введите класс получаемых объектов: ')
            cls: OntologyObject.__class__ | None = get_class(class_name)
            if cls is None:
                print('Такого класса не существует\n\n')
            else:
                break

        while True:
            instance = get_random_class_instance(cls)
            available_fields = [
                key for key in instance.__dict__.keys()
                if not re.match(r"__\w*__", key)
            ]
            field = input(
                f'Введите требуемое поле ( {", ".join(available_fields)} ): ')
            try:
                getattr(instance, field)
                break
            except Exception:
                print('Такого поля в классе не существует\n\n')

        value = input('Введите значение поля: ')
        res = find_related_objects_by_value(cls, field, value)

        while True:
            flag = False
            if res is None or len(res) == 0:
                print('Объекты по введённому запросу не найдены')
                while True:
                    _type = input(
                                '1 - повторить ввод запроса\nq - завершить
выполнение программы\n'
                    )
                    if _type == 1:
                        break
                    elif _type == 'q':
                        exit(0)
            else:
                str_objects = [
                    "\033[32m" + str(obj) + "\033[0m (\033[33m" +
                    str(obj_type) + "\033[0m)" for obj, obj_type in res
                ]
                print(f'Полученные объекты: {", ".join(str_objects)}')
                while True:
                    _type = input('1 - повторить ввод запроса\n'
                                  '2 - посмотреть связанные объекты\n'
                                  'q - завершить выполнение программы\n')
                    if _type == '1':
```

Окончание Листинга А.1

```
                        flag = False
                        break
                    elif _type == '2':
                        flag = True
                        break
                    elif _type == 'q':
                        exit(0)

                if flag:
                    if len(res) == 1:
                        obj_number = 1
                    else:
                        while True:
                            obj_number = int(
                                input(
                                        f'Выберите номер нужного объекта (1-
{len(res)}): '
                                ))
                            if obj_number not in range(1, len(res) + 1):
                                print('Введён неправильный номер')
                            else:
                                break

                    instance = res[obj_number - 1][0]
                    related_classes = get_related_class(instance)
                    res = []
                    for rel_class, field_name in related_classes:
                        res.extend(
                            find_related_objects_by_value(
                                rel_class, field_name, instance))
                else:
                    break
            if not flag:
                break


main()
```

Приложение Б

Код реализации метода имитации отжига в задаче коммивояжёра на языке Python

Листинг Б.1 — Код файла generate.py

```
import pandas as pd
import numpy as np


def generate(min_coord: int, max_coord: int, n_places: int):
    coord_x = np.random.randint(min_coord, max_coord, size=(n_places, ))
    coord_y = np.random.randint(min_coord, max_coord, size=(n_places, ))
    return coord_x, coord_y


def main():
    min_coord, max_coord, n = -100, 100, 10
    x, y = generate(min_coord, max_coord, n)
    df = pd.DataFrame({'x': x, 'y': y})
    df.to_csv('data.csv', index=False)


if __name__ == '__main__':
    main()
```

Листинг Б.2 — Код файла data.csv

```
x,y
3,-85
-88,-5
-30,-79
-100,14
25,76
36,61
3,91
15,96
-66,-54
-81,77
```

Листинг Б.3 — Код файла main.py

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import random


class Path:
    graph: np.ndarray | None = None
    x: np.ndarray | None = None
    y: np.ndarray | None = None

    def __init__(self,
                 path: list[int] | None = None,
                 graph: np.ndarray | None = None,
                 x: np.ndarray | None = None,
                 y: np.ndarray | None = None):
        if path is None:
            self.path = [0]
        else:
            self.path = path

        if Path.graph is None:
            Path.graph = graph
        if Path.x is None:
            Path.x = x
        if Path.y is None:
            Path.y = y
```

Продолжение Листинга Б.3

```
        if Path.graph is None:
            raise Exception('Ошибка с данными о графе')

    def create_new_path(self):
        while not all([x in self.path for x in range(Path.graph.shape[0])]):
            possible_places = [
                i for i in range(Path.graph.shape[0])
                if i not in self.path
            ]
            next_place = random.choice(possible_places)
            self.path.append(next_place)
        self.path.append(0)

    def get_replacements(self):
        possible_replacements = [
            self.path[i] for i in range(1, len(self.path))
            if self.path[i] != 0
        ]
        while True:
            first, second = random.choices(possible_replacements, k=2)
            if first != second:
                break
        first_index, second_index = self.path.index(first), self.path.index(
            second)
        if first_index > second_index:
            first_index, second_index = second_index, first_index
            first, second = second, first
        return first_index, second_index

    def recreate_path(self):
        first_index, second_index = self.get_replacements()
        self.path = [elem for elem in self.path]
      self.path[first_index], self.path[second_index] = self.path[second_index],
self.path[first_index]
        print(f'Замены: {self.path[first_index]}, {self.path[second_index]}')

    @property
    def length(self) -> float:
        length = 0
        for i in range(1, len(self.path)):
            length += Path.graph[self.path[i]][self.path[i - 1]]
        return length

    def __str__(self) -> str:
        return ', '.join(map(str, self.path))

    def print_verbose(self):
        result_str = ''
        for i, point in enumerate(self.path):
            result_str += str(point)

            if i != len(self.path) - 1:
                result_str += ' -> '
        return result_str

    def print_length(self):
        cum_length = []
        for i in range(1, len(self.path)):
            edge_len = Path.graph[self.path[i]][self.path[i - 1]]
            if not cum_length:
                cum_length.append(edge_len)
            else:
                cum_length.append(edge_len)
        return " + ".join(map(lambda x: str(round(x, 2)), cum_length))

    def draw_path(self):
        for i in range(len(Path.graph)):
            plt.text(Path.x[i], Path.y[i], f'{i}')
        for v0, v1 in zip(self.path, self.path[1:]):
            x = (Path.x[v0], Path.x[v1])
```

Продолжение Листинга Б.3

```
            y = (Path.y[v0], Path.y[v1])
            plt.plot(x, y)
        plt.show()


class Solution:

    def __init__(self, file_path):
        self.file_path = file_path
        self.current_path: Path
        self.best_path: Path | None = None

    def create_graph(self):
        df = pd.read_csv(self.file_path)
        # df = pd.concat([pd.DataFrame([[0, 0, 0]], columns=df.columns), df],
        #                ignore_index=True)
        x = df['x'].to_numpy()
        y = df['y'].to_numpy()
        graph = np.ndarray(shape=(len(df), len(df)))
        for i in range(len(df)):
            for j in range(len(df)):
                graph[i][j] = np.sqrt((x[i] - x[j])**2 + (y[i] - y[j])**2)
                if i < j:
                    sign_1 = '-' if x[j] >= 0 else '+'
                    sign_2 = '-' if y[j] >= 0 else '+'
                        print(f'sqrt(({x[i]} {sign_1} {abs(x[j])})^2 + ({y[i]}
{sign_2} {abs(y[j])})^2) = {round(graph[i][j], 2)} \\')

        for i in range(len(df)):
            for j in range(len(df)):
                if i < j:
                    print(f'[{i} #math.arrow {j}], [{graph[i][j]: .2f}],')

        self.current_path = Path(graph=graph, x=x, y=y)
        self.current_path.create_new_path()

    def solve(self):
        temperature = 1000
        history = []
        self.best_path = self.current_path
        history.append(self.best_path.length)
        print('Начальный путь: ' + self.best_path.print_verbose())
            print(f'Длина начального пути: {self.best_path.print_length()} =
{self.best_path.length:.2f} (м)')
        print('============\n')
        i = 0
        while temperature > 1:
            i += 1
            print(f'Температура: {temperature}')
            print('=============')
            self.current_path = Path(self.best_path.path)
            self.current_path.recreate_path()
            print('Лучший путь: ' + self.best_path.print_verbose())
            print(f'Длина лучшего пути: {self.best_path.length:.2f} м')
            print('Текущий путь: ' + self.current_path.print_verbose())
             print(f'Длина текущего пути: {self.current_path.print_length()} =
{self.current_path.length:.2f} (м)')
            if self.current_path.length < self.best_path.length:
                self.best_path = self.current_path
            else:
                prob_lim = np.exp(-(self.current_path.length -
                                    self.best_path.length) /
                                  (temperature))
                probability = np.random.random()
                print(f'H = {prob_lim}; p = {probability}')
                if probability < prob_lim:
                    self.best_path = self.current_path
            temperature *= 0.9
            print('==============\n')
            history.append(self.best_path.length)
```

Окончание Листинга Б.3

```
        print('Найденное лучшее решение: ' + self.best_path.print_verbose())
        print(f'Длина найденного решения: {self.best_path.length:.2f} m')
        plt.plot(history)
        plt.show()
        self.best_path.draw_path()


def main():
    solution = Solution('backup_6.csv')
    solution.create_graph()
    solution.solve()


if __name__ == '__main__':
    main()
```

Приложение В

Код реализации метода имитации отжига в задаче оптимизации функции на языке Python

Листинг В.1 — Реализация метода имитации отжига

```
import numpy as np
import matplotlib.pyplot as plt


class Solution:

    def __init__(self):
        self.current_solution: np.ndarray
        self.best_solution: np.ndarray | None = None
        self._max = 5.12
        self._min = -self._max
        self.n = 2

    @staticmethod
    def rastrigin(x: np.ndarray):
        return 10 * len(x) + np.sum(x**2 - 10 * np.cos(2 * np.pi * x))

    def init_solution(self):
      self.best_solution = self.current_solution = np.random.random(size=self.n)
* (self._max - self._min) + self._min

    @staticmethod
    def cauchy_distribution(x, main_x, temperature):
         return ((1 / np.pi) * temperature / ((x - main_x) ** 2 + temperature
** 2))

    def generate_solution(self, temperature: float):
        while True:
              new_x = np.random.random(size=self.n) * (self._max - self._min)
+ self._min
               p_distribute = Solution.cauchy_distribution(self.best_solution,
new_x, temperature)
            p = np.random.random(size=p_distribute.shape)
            if np.all(p <= p_distribute):
                return new_x

    def solve(self):
        temperature = 10
        t0 = temperature
        history = []
        history.append(self.rastrigin(self.best_solution))
        print(
              f'Исходное решение: ({", ".join(map(lambda x: str(round(x, 2)),
self.current_solution))})'
        )
        print(
          f'Исходное значение функции: {self.rastrigin(self.current_solution)}')
        print('============\n')
        k = 2
        while t0 > 0.1:
            print(f'Температура: {t0}')
            print('=============')
            self.current_solution = self.generate_solution(temperature)
            current_rastrigin, best_rastrigin = self.rastrigin(
                self.current_solution), self.rastrigin(self.best_solution)
            print(
                    f'Лучшее решение: ({", ".join(map(lambda x: str(round(x,
2)), self.best_solution))})')
            print(f'Лучшее значение функции: {best_rastrigin}')
            print(
                  f'Текущее решение: ({", ".join(map(lambda x: str(round(x,
2)), self.current_solution))})'
```

Окончание Листинга В.1

```
            )
            print(f'Текущее значение функции: {current_rastrigin}')
            if current_rastrigin < best_rastrigin:
                self.best_solution = self.current_solution
            else:
                prob_lim = np.exp(-(current_rastrigin - best_rastrigin) / t0)
                probability = np.random.random()
                print(f'H = {prob_lim}; p = {probability}')
                if probability < prob_lim:
                    self.best_solution = self.current_solution
            t0 = temperature / (k ** (1 / self.n))
            k += 1
            print('===============\n')
            history.append(self.rastrigin(self.best_solution))

        best_rastrigin = self.rastrigin(self.best_solution)
          print(f'Лучшее решение: ({", ".join(map(lambda x: str(round(x, 2)),
self.best_solution))})')
        print(f'Лучшее значение функции: {best_rastrigin}')
        plt.plot(history)
        plt.show()


def main():
    solution = Solution()
    solution.init_solution()
    solution.solve()


if __name__ == '__main__':
    main()
```

Приложение Г

Реализация роевого алгоритма в задаче оптимизации на языке Python

Листинг Г.1 — Реализация роевого алгоритма

```
import numpy as np
import matplotlib.pyplot as plt


def rastrigin(x: np.ndarray):
    return 10 * len(x) + np.sum(x**2 - 10 * np.cos(2 * np.pi * x))


class Particle:
    def __init__(self):
        self._max = 5.12
        self._min = -self._max
        self.n = 2
            self.x = np.random.random(size=self.n) * (self._max - self._min)
+ self._min
        self.best_x = self.x.copy()
        self.speed = np.zeros(shape=(self.n,))

    def correct_speed(self, global_best: np.ndarray):
        c1 = 2
        c2 = 2
        alpha = np.random.random()
        self.speed += c1 * alpha * (self.best_x - self.x) + c2 * (1 - alpha) *
(global_best - self.x)

    def correct_position(self):
        self.x += self.speed
        for i in range(len(self.x)):
            if self.x[i] > self._max:
                self.x[i] = self._max
            elif self.x[i] < self._min:
                self.x[i] = self._min

    def __str__(self):
        return f"Текущая позиция: {self.x} -- лучшая позиция: {self.best_x}"


class Swarm:
    def __init__(self, particle_count: int = 10):
        self.particles = [Particle() for _ in range(particle_count)]
        self.best_solution: np.ndarray | None = None

    def solution_step(self):
        if self.best_solution is None:
            self.best_solution = self.particles[0].best_x.copy()

        for particle in self.particles:
            if rastrigin(particle.x) < rastrigin(particle.best_x):
                particle.best_x = particle.x.copy()
            if rastrigin(particle.best_x) < rastrigin(self.best_solution):
                self.best_solution = particle.best_x.copy()

        for particle in self.particles:
            particle.correct_speed(self.best_solution)
            particle.correct_position()

        return rastrigin(self.best_solution), self.best_solution

    def draw_swarm(self):
        particle_x = [particle.x[0] for particle in self.particles]
        particle_y = [particle.x[1] for particle in self.particles]
        ax = plt.gca()
        ax.set_xlim(-5.12, 5.12)
        ax.set_ylim(-5.12, 5.12)
        plt.scatter(particle_x, particle_y, s=[1.5 for _ in self.particles])
```

Окончание Листинга Г.1

```
        plt.show()


class Solution:

    def __init__(self):
        self.swarm = Swarm(particle_count=1000)

    def solve(self):
        steps = 50
        history = []
        for step in range(1, steps + 1):
            value, x = self.swarm.solution_step()
            print(f'Итерация: {step}. Значение: {value} в точке {x}')
            history.append(value)
        plt.plot(history)
        plt.show()


def main():
    solution = Solution()
    solution.solve()


if __name__ == '__main__':
    main()
```

Приложение Д

Реализация муравьиного алгоритма на языке Python

Листинг Д.1 — Реализация муравьиного алгоритма

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import random
import time
from math import sqrt


class GraphElement:
    def __init__(self, distance: float | None = None, pheromone: float = 0):
        self.distance = distance
        self.pheromone = pheromone


class Path:
    graph: list[list[GraphElement]] | None = None
    x: np.ndarray | None = None
    y: np.ndarray | None = None

    def __init__(self,
                 path: list[int] | None = None,
                 graph: list[list[GraphElement]] | None = None,
                 x: np.ndarray | None = None,
                 y: np.ndarray | None = None):
        if path is None:
            self.path = [0]
        else:
            self.path = path

        if Path.graph is None:
            Path.graph = graph
        if Path.x is None:
            Path.x = x
        if Path.y is None:
            Path.y = y

        if Path.graph is None:
            raise Exception('Граф не построен')

     def create_new_path(self, alpha: float, beta: float, start_position: int
= 0):
        self.path = [start_position]
        while not all([x in self.path for x in range(len(Path.graph))]):
            current_place = self.path[-1]
            possible_places = [
                i for i in range(len(Path.graph))
                if i not in self.path
            ]
            pheromones = []
            for next_place in possible_places:
             pheromones.append((Path.graph[current_place][next_place].pheromone
** alpha
                                                * (1 / Path.graph[current_place]
[next_place].distance) ** beta))
            total_pheromone = sum(pheromones)
            pheromones = [elem / total_pheromone for elem in pheromones]

            probability = np.random.random()
            total = 0
            for next_place, ph in zip(possible_places, pheromones):
                total += ph
                if probability <= total:
                    self.path.append(next_place)
                    break
```

Продолжение Листинга Д.1

```
        self.path.append(self.path[0])

    @property
    def length(self) -> float:
        length = 0
        for i in range(1, len(self.path)):
            length += Path.graph[self.path[i]][self.path[i - 1]].distance
        return length

    def __str__(self) -> str:
        return ', '.join(map(str, self.path))

    def print_verbose(self, start_at_zero: bool = False):
        result_str = ''
        if start_at_zero:
            if self.path.index(0) != 0:
                index = self.path.index(0)
                first_part = self.path[index:]
                second_part = self.path[1:index]
                path = first_part + second_part + [0]
            else:
                path = self.path
        else:
            path = self.path
        for i, point in enumerate(path):
            result_str += str(point)

            if i != len(path) - 1:
                result_str += ' -> '
        return result_str

    def print_length(self):
        cum_length = []
        for i in range(1, len(self.path)):
            edge_len = Path.graph[self.path[i]][self.path[i - 1]].distance
            if not cum_length:
                cum_length.append(edge_len)
            else:
                cum_length.append(edge_len)
        return " + ".join(map(lambda x: str(round(x, 2)), cum_length))

    def draw_path(self):
        for i in range(len(Path.graph)):
            plt.text(Path.x[i], Path.y[i], f'{i}')
        for v0, v1 in zip(self.path, self.path[1:]):
            x = (Path.x[v0], Path.x[v1])
            y = (Path.y[v0], Path.y[v1])
            plt.plot(x, y)
        plt.show()


class Ant:
    def __init__(self):
        self.path = Path()


class AntColony:
    def __init__(self, ant_count: int = 10):
        self.ants = [Path() for _ in range(ant_count)]
        self.vapor_rate = 0.95
        self.alpha = 6
        self.beta = 7

    def solution_step(self):
        for i, ant in enumerate(self.ants):
            ant.create_new_path(self.alpha, self.beta, i)
            # print(ant.print_verbose())

        for i in range(len(Path.graph)):
            for j in range(len(Path.graph)):
                if i != j:
```

Продолжение Листинга Д.1

```
                    Path.graph[i][j].pheromone *= (1 - self.vapor_rate)

           delta_pheromones = [[0 for _ in range(len(Path.graph))] for _ in
range(len(Path.graph))]
        for ant in self.ants:
            delta_pheromone = 100 / ant.length
            for i, j in zip(ant.path, ant.path[1:]):
                delta_pheromones[i][j] += delta_pheromone

        for i in range(len(Path.graph)):
            for j in range(len(Path.graph)):
                Path.graph[i][j].pheromone += delta_pheromones[i][j]


        best_path = self.ants[0]
        best_length = best_path.length
        for ant in self.ants[1:]:
            if ant.length < best_length:
                best_path = ant
                best_length = best_path.length

        return Path(best_path.path)


class Solution:

    def __init__(self, file_path):
        self.file_path = file_path
        self.best_path: Path | None = None

    def create_graph(self):
        df = pd.read_csv(self.file_path)
        x = df['x'].to_numpy()
        y = df['y'].to_numpy()
         self.graph = graph = [[GraphElement() for _ in range(len(df))] for _
in range(len(df))]
        pheromone = 0.78498
        for i in range(len(df)):
            for j in range(i, len(df)):
                    graph[i][j].distance = graph[j][i].distance = sqrt((x[i] -
x[j])**2 + (y[i] - y[j])**2)
                if i != j:
                    graph[i][j].pheromone = graph[j][i].pheromone = pheromone
        Path(graph=graph, x=x, y=y)

    def solve(self):
        ant_colony = AntColony(ant_count=6)
        steps = 40
        history = []
        mean_history = []
        for step in range(steps):
            print(f'Итерация {step + 1} / {steps}')
            if (step + 1) % (steps // 5) == 0 or step == 0:
                for ant in ant_colony.ants:
                    print(ant.print_verbose(start_at_zero=True))
                    print(ant.length)
            current_path = ant_colony.solution_step()
            mean = 0
            for ant in ant_colony.ants:
                mean += ant.length
            mean /= len(ant_colony.ants)
            mean_history.append(mean)
                        if self.best_path is None or current_path.length <
self.best_path.length:
                self.best_path = current_path
            history.append(self.best_path.length)
                                                        print(f'Лучший    путь:
{self.best_path.print_verbose(start_at_zero=True)}')
            print(f'Длина лучшего пути: {self.best_path.length}')
            print('\n\n\n')
        fig, ax = plt.subplots()
```

Окончание Листинга Д.1

```
        ax.set_title(f'Длина лучшего пути: {self.best_path.length:.2f}')
        plt.plot(history, label='Длина лучшего пути')
        plt.plot(mean_history, label='Средняя длина пути среди муравьёв')
        plt.legend(loc='upper right')
        plt.show()
        self.best_path.draw_path()



def main():
    solution = Solution('../prac2/backup_10.csv')
    solution.create_graph()
    solution.solve()


if __name__ == '__main__':
    main()
```

Приложение Е

Реализация алгоритма пчелиной колонии на языке Python

Листинг Е.1 — Реализация алгоритма пчелиной колонии

```
import numpy as np
import matplotlib.pyplot as plt


def rastrigin(x: np.ndarray):
    return 10 * len(x) + np.sum(x**2 - 10 * np.cos(2 * np.pi * x))


class BeeColony:

    def __init__(self,
                 scout_bee_count: int = 10,
                 optimal_bee_count: int = 5,
                 suboptimal_bee_count: int = 2,
                 optimal_solution_count: int = 2,
                 suboptimal_solution_count: int = 3,
                 field_size: float = 1.5):
        self.scout_bee_count = scout_bee_count
        self.optimal_bee_count = optimal_bee_count
        self.suboptimal_bee_count = suboptimal_bee_count
        self.optimal_solution_count = optimal_solution_count
        self.suboptimal_solution_count = suboptimal_solution_count
        self.field_size = field_size

        self._max = 5.12
        self._min = -self._max
        self.n = 2

    def solution_step(self, previous_result: np.ndarray | None = None):
        if previous_result is None:
            bee_area = np.random.random(size=(self.scout_bee_count, self.n)
                                        ) * (self._max - self._min) + self._min
        else:
            bee_area = previous_result
        function_values = np.array([rastrigin(x) for x in bee_area])
        bee_area = bee_area[function_values.argsort()]

        optimal_solution = bee_area[:self.optimal_solution_count]
        suboptimal_solution = bee_area[self.optimal_solution_count:self.
                                       optimal_solution_count +
                                       self.suboptimal_solution_count]

        new_bee_area = []
        for solution in optimal_solution:
            min_search_field = solution - self.field_size
            max_search_field = solution + self.field_size
            new_bee_area.append(solution)
            for _ in range(self.optimal_bee_count - 1):
                bee = np.random.random(self.n) * (
                    max_search_field - min_search_field) + min_search_field
                new_bee_area.append(bee)

        for solution in suboptimal_solution:
            min_search_field = solution - self.field_size
            max_search_field = solution + self.field_size
            new_bee_area.append(solution)
            for _ in range(self.suboptimal_bee_count - 1):
                bee = np.random.random(self.n) * (
                    max_search_field - min_search_field) + min_search_field
                new_bee_area.append(bee)
        return np.array(new_bee_area)


class Solution:
```

Окончание Листинга Е.1

```
    def __init__(self):
        self.bee_colony = BeeColony(scout_bee_count=100,
                                    optimal_bee_count=30,
                                    suboptimal_bee_count=10,
                                    optimal_solution_count=10,
                                    suboptimal_solution_count=5,
                                    field_size=0.5)

    def solve(self):
        history = []
        result: np.ndarray | None = None
        best_result_value = float("inf")
        best_result_repeat = 0
        while result is None or best_result_repeat < 1000:
            result = self.bee_colony.solution_step(result)
            function_values = np.array([rastrigin(x) for x in result])
            if np.min(function_values) < best_result_value:
                best_result_value = np.min(function_values)
                best_result = result[function_values.argmin()]
                best_result_repeat = 0
                history.append(best_result_value)
            else:
                best_result_repeat += 1
            print(
                f'Лучший результат {best_result_value} в точке {best_result}.'
                f' Количество повторений: {best_result_repeat}')
        plt.plot(history)
        plt.show()


def main():
    solution = Solution()
    solution.solve()


if __name__ == '__main__':
    main()
```

Приложение Ж

Реализация электромагнитного алгоритма на языке Python

Листинг Ж.1 — Реализация электромагнитного алгоритма

```
import numpy as np
import matplotlib.pyplot as plt


def rastrigin(x: np.ndarray):
    return np.sum(x**2 - 10 * np.cos(2 * np.pi * x) + 10)


class EMA:
    def __init__(self, n: int):
        self.n = n
        self.population_size = 10 * n
        self.max_iter = 50 * n
        self.local_iter = 10
        self.scale = 0.005
        self._min = -5.12
        self._max = -self._min

    def calculate_best(self):
        self.values = np.array([rastrigin(x) for x in self.x])
        self.best_value = np.min(self.values)
           self.best_x = self.x[np.where(abs(self.values - self.best_value) <
1e-3)].flatten()

    def create_population(self):
        self.x = np.vstack([self._min + np.random.uniform(0, 1, size=self.n) *
(self._max - self._min)
                            for _ in range(self.population_size)])
        self.calculate_best()

    def local_search(self):
        search_field = self.scale * (self._max - self._min)
        for k, particle in enumerate(self.x):
            cnt = 0
            while cnt < self.local_iter:
                for i in range(self.n):
                    sign = np.random.randint(0, 2) * 2 - 1
                    y = particle.copy()
                    velocity = np.random.uniform()
                    y[i] += sign * velocity * search_field
                    if rastrigin(y) < rastrigin(particle):
                        self.x[k] = y.copy()
                        cnt = self.local_iter
                        break
                    cnt += 1
        self.calculate_best()

    def calculate_force(self):
            self.q = np.exp(-self.n * (self.values - self.best_value) /
(np.sum(self.values - self.best_value)))
        self.force = np.zeros_like(self.x)
        for i in range(self.population_size):
            for j in range(self.population_size):
                if i != j:
                    if self.values[j] < self.values[i]:
                                self.force[i] += ((self.x[j] - self.x[i]) /
np.linalg.norm(self.x[j] - self.x[i]) ** 2) \
                                         * self.q[i] * self.q[j]
                    else:
                                self.force[i] += ((self.x[i] - self.x[j]) /
np.linalg.norm(self.x[j] - self.x[i]) ** 2) \
                                         * self.q[i] * self.q[j]

    def move_particles(self):
        for i in range(self.population_size):
```

Окончание Листинга Ж.1

```
            if abs(self.values[i] - self.best_value) > 1e-3:
                alpha = np.random.uniform()
                velocity = np.ones_like(self.x[i])
               normalized_force = self.force[i] / np.linalg.norm(self.force[i])
                for j in range(self.n):
                    if self.force[i][j] > 0:
                        velocity[j] = self._max - self.x[i][j]
                    else:
                        velocity[j] = self.x[i][j] - self._min
                self.x[i] += alpha * np.multiply(normalized_force, velocity)

    def solve(self):
        self.create_population()
        history = []
        for i in range(self.max_iter):
            history.append(self.best_value)
           print(f'Текущее лучшее значение: {round(self.best_value, 4)} в точке
{list(map(lambda x: round(x, 4), self.best_x))}')
            print(f'Итерация: {i + 1}')
            self.local_search()
            self.calculate_best()
            self.calculate_force()
            self.move_particles()
        plt.plot(history)
        plt.show()



def main():
    ema = EMA(2)
    ema.solve()


if __name__ == '__main__':
    main()
```