

Санкт-Петербургский государственный университет
Прикладная математика и информатика

Отчет по учебной практике 2 (научно-исследовательской работе) (семестр 3)

МАТЕМАТИЧЕСКОЕ МОДЕЛИРОВАНИЕ ДЛЯ РЕШЕНИЯ
МНОГОКРИТЕРИАЛЬНЫХ ОПТИМИЗАЦИОННЫХ ЗАДАЧ

Выполнил:

Мажара Евгений Николаевич

группа 20.Б06-мм

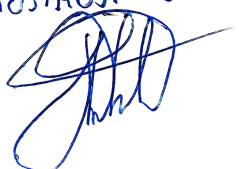


Научный руководитель:

к. ф.-м. н., доцент

Шпилев Петр Вальерьевич

Работа выполнена
в полном объеме



Отметка о зачете:

<< Работа выполнена на высоком уровне и может быть зачтена с оценкой А >>

Отзыв на учебную практику 2 (научно – исследовательскую работу) студента 2-го курса бакалавриата Мажара Евгения Николаевича

Работа посвящена изучению принципов математического моделирования. В рамках данной работы студент самостоятельно ознакомился с алгоритмом роя частиц. На языке Java были написаны программные реализации канонического варианта данного алгоритма и его различных модификаций. Работа алгоритма проиллюстрирована на примерах многочисленных тестовых функций.

Работа написана аккуратно, поставленная задача реализована в полном объеме. Считаю, что работа может быть зачтена с оценкой А.

28.12.21



Петр Валерьевич Шпилев

Оглавление

Глава 1. Вступление	4
1.1. Введение	4
1.2. Классификация оптимизируемых функций	6
1.3. Классификация популяционных алгоритмов задачи многомерной оптимизации	9
1.4. Постановка задачи	11
Глава 2. Алгоритм роя частиц	12
2.1. Предыстория	12
2.2. Описание канонического алгоритма	13
2.3. Заполнение множества индексов соседей. Топологии соседства частиц	17
Глава 3. Программная реализация алгоритма	22
3.1. Класс FreeParameters	24
3.2. Класс Vector	27
3.3. Класс Agent	28
3.4. Класс Swarm	32
3.5. Интерфейс Topology	34
3.6. Класс RingTopology	34
3.7. Класс CliqueTopology	35
3.8. Класс TorusTopology	35
3.9. Класс ClasterTopology	36
3.10. Класс StartAlgorithm	37
3.11. Класс MultiStart	39
3.12. Класс Interface	41
Глава 4. Тестирование работы алгоритмов	42
4.1. Значения свободных параметров	42
4.2. Критерии оценивания алгоритма	43
4.3. Функция сферы	44
4.4. Функция Дэвиса	47

4.5.	Функция Швевеля	49
4.6.	Функция Экли	52
4.7.	Функция Растригина	55
4.8.	Функция Розенброка	58
4.9.	Многоэкстремальная функция	61
4.10.	Полином, имеющий несколько локальных минимумов	64
4.11.	Функция Гриванка	67
4.12.	Вывод	70
Глава 5.	Заключение	72

Введение

Достаточно часто задачи, возникающие в таких науках как физика, химия, молекулярная биология и во многих других, сводятся к **оптимизации некоторой многокритериальной функции**. При этом такие функции имеют характерные особенности: нелинейность, недифференцируемость, многоэкстремальность, овраженность, плохая формализованность, высокая вычислительная сложность, высокая размерность, сложная топология пространства поиска и другие. Именно эти свойства задач глобальной оптимизации объясняют отсутствие универсального алгоритма решения и в то же время наличие большого числа алгоритмов оптимизации, каждый из которых ориентирован на тот или иной тип функций.

Для эффективного решения задач глобальной оптимизации был разработан класс стохастических поисковых алгоритмов, получивших название **популяционных алгоритмов**. Эти алгоритмы являются альтернативой траекторным алгоритмам, однако в любом популяционном алгоритме в области поиска эволюционируют сразу несколько кандидатов на решение задачи. Все популяционные алгоритмы относятся к классу эвристических алгоритмов, то есть их сходимость к глобальному решению не доказана, однако экспериментально установлено, что в подавляющем числе случаев сходимость выполняется.

Популяционные алгоритмы основаны на математических моделях поведения популяций живых существ, например, колоний пчел, муравьев, а также поведение птиц при полете в стае. Кроме того, некоторые популяционные алгоритмы, например, генетический алгоритм, используют эволюцию живых существ как источник инспирирования. Для обозначения членов популяции часто используются такие термины и названия как индивид, пчела, муравей, особь и т.д. В данной работе будут использованы термины **агент** или **частица**. Как в живой природе, так и в алгоритмах популяции агентов присущи некоторые свойства:

- **Автономность** - каждый агент движется в пространстве поиска хотя бы частично независимо друг от друга
- **Ограниченность представления** - каждый агент имеет представление лишь о некоторой области пространства поиска и, возможно, об окружении некоторых других агентов

- **Коммуникабельность** - агенты имеют возможность обмениваться информацией о топологии исследуемой функции
- **Децентрализованность** - отсутствие центра, то есть агента, управляющего процессом поиска в целом

Именно эти свойства популяции обеспечивают формирование ее **роевого интеллекта** (*swarm intelligence*), который проявляется в сложном поведении популяции в целом при достаточно простом поведении каждого агента.

Одним из самых распространенных популяционных алгоритмов является **алгоритм роя частиц** (*PSO – particle swarm optimization*). Он относится к популяционным алгоритмам, вдохновленным живой природой. В основу этого алгоритма легла социально-психологическая поведенческая модель толпы. Существует множество различных модификаций этого алгоритма, вдохновленных такими явлениями природы как полетом стаи птиц и передвижением рыб в косяке. В данной работе будет продемонстрирован принцип работы канонического алгоритма *PSO*, а также его программная реализация на языке Java.

Классификация оптимизируемых функций

Существует немалое количество классификаций задач оптимизации по различным признакам.

- **Характер ограничений на область поиска:**

Если пространство поиска задано в виде

$$D = \{X \in \mathbb{R}^{|X|} : E(X) = 0, G(X) \geq 0\},$$

то задача называется задачей с ограничениями общего вида.

Если пространство поиска задано в виде

$$D = \{X \in \mathbb{R}^{|X|} : E(X) = 0\},$$

то задача носит название задачи с ограничениями типа равенств.

Аналогично, если пространство поиска задано в виде

$$D = \{X \in \mathbb{R}^{|X|} : G(X) \geq 0\},$$

то задача является задачей с ограничениями типа неравенств.

Наконец, если пространство поиска имеет вид

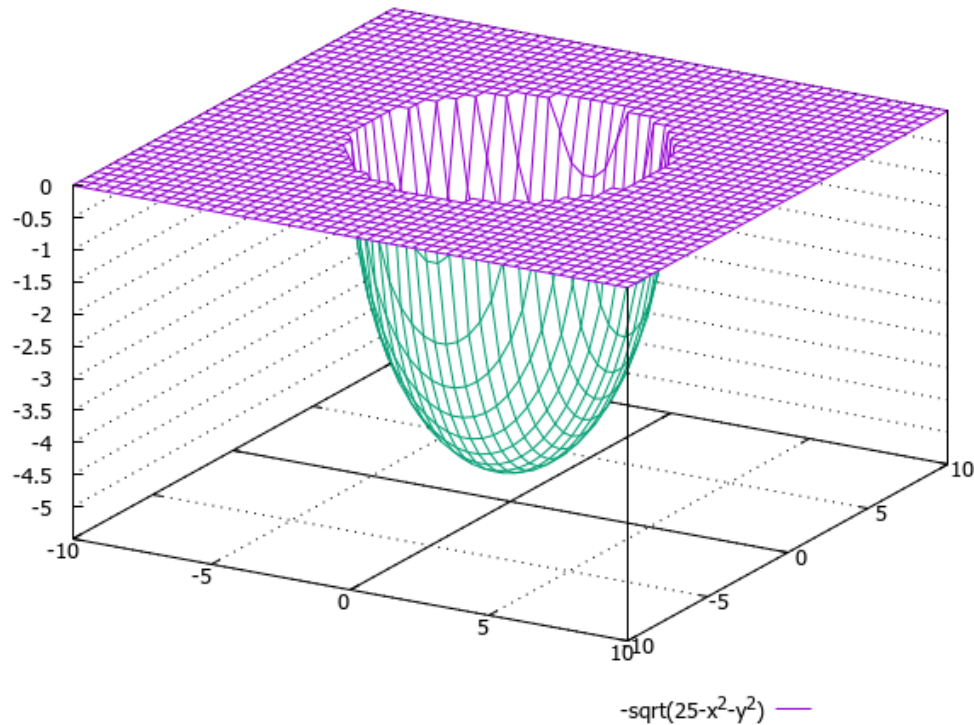
$$D = \{X \in \mathbb{R}^{|X|}\},$$

то задача называется задачей оптимизации без ограничений или задачей безусловной оптимизации.

Если же на пространство поиска наложены какие-либо ограничения, задача носит название задачи условной оптимизации.

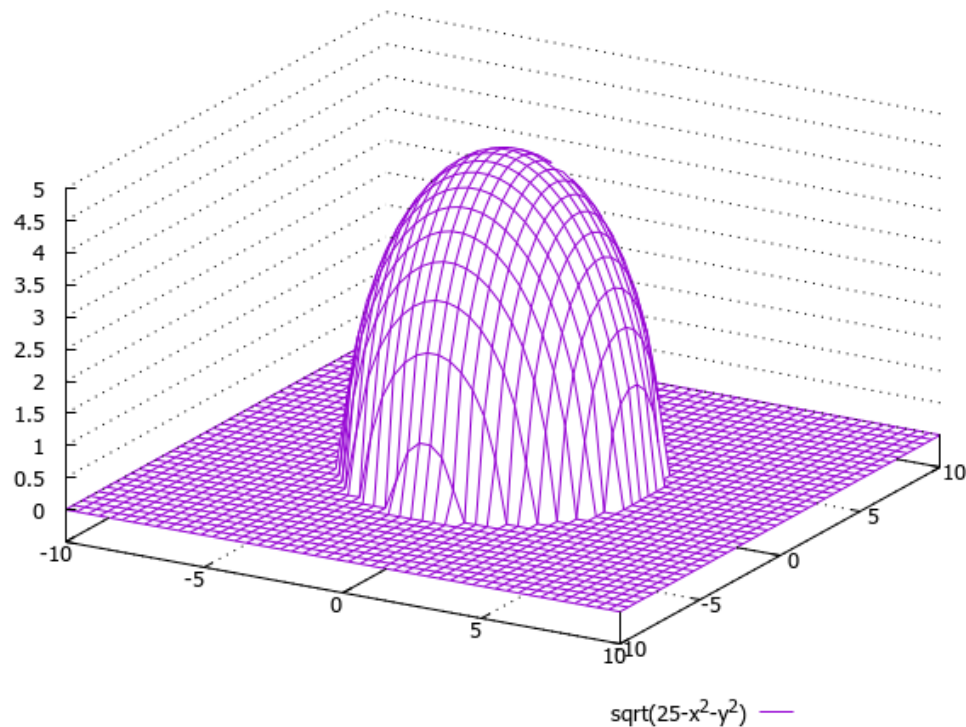
- **Число точек минимума исследуемой функции:**

Если целевая функция f имеет в заданной области D один минимум, то задача является **задачей одноэкстремальной оптимизации.**



пример одноэкстремальной функции: $f(x, y) = -\sqrt{25 - x^2 - y^2}$, $f^* = f(0, 0) = -5$

Если же целевая функция f имеет более одного минимума в заданной области поиска D , то задача называется **задачей многоэкстремальной оптимизации.**



пример многоэкстремальной функции: $f(x, y) = \sqrt{25 - x^2 - y^2}$, $f^* = f((x, y) : x^2 + y^2 = 25) = 0$

- **Размерность вектора варьируемых параметров:**

Если размерность вектора X $|X| = 1$, то задача носит название **однопараметрической задачи оптимизации**.

Если размерность вектора X $|X| > 1$, то задача носит название **многопараметрической** или **многомерной задачи оптимизации**.

- **Характер искомого решения:**

Если задача состоит в поиске любого локального минимума в области D , то задача называется **задачей локальной оптимизации**.

В случае же когда целью является поиск глобального минимума целевой функции задача называется **задачей глобальной оптимизации**.

Классификация популяционных алгоритмов задачи многомерной оптимизации

Алгоритмы, используемые для решения задач оптимизации целевой функции также могут быть классифицированы по разным свойствам.

Прежде всего алгоритмы классифицируются по аналогичным функциям свойствам:

- **Характер искомого решения:**

В зависимости от типа поиска минимумов, на которые нацелен алгоритм, его по аналогии с задачами оптимизации называют **алгоритмом локальной либо глобальной оптимизации**. Ясно, что каждый алгоритм применяется к соответствующей задаче оптимизации.

- **Характер ограничений на область поиска:**

В зависимости от того, на какой тип задач ориентирован алгоритм, он называется **алгоритмом безусловной либо условной оптимизации**. Ясно, что каждый алгоритм применяется к соответствующей задаче оптимизации.

Существуют также и независимые от типа рассматриваемой целевой функции свойства:

- **Характер функции, вычисляющей следующие положения каждого агента:**

Если функция Φ , которая по заданному правилу вычисляет новые значения вектора варьируемых параметров X , является детерминированной, то есть не содержит параметров, принимающих случайные значения, то алгоритм носит название **детерминированного алгоритма оптимизации**.

В ином случае, когда функция Φ , вычисляющая новые положения агентов, содержит случайные компоненты, алгоритм называется стохастическим алгоритмом оптимизации.

- **Число предыдущих учитываемых шагов алгоритма:**

В ситуации, когда функция Φ , вычисляющая новые положения агентов, использует только информацию об одной предыдущей итерации алгоритма, алгоритм является одношаговым алгоритмом оптимизации.

Если же функция Φ при вычислении новых положений использует информацию о нескольких предыдущих итерациях, то алгоритм носит название **многошагового алгоритма оптимизации** либо алгоритма с памятью.

Также важными свойствами любого алгоритма являются интенсивность и диверсификация поиска.

- Интенсивность поиска показывает, насколько быстро алгоритм сходится, то есть завершает свою работу
- Диверсификация поиска показывает, насколько широко осуществляется поиск в данной области D

В любом алгоритме оптимизации важен баланс между этими двумя показателями. С одной стороны, поиск завершается тем быстрее, чем менее широко исследуется область поиска, с другой стороны, при слишком узком исследовании есть вероятность определения локального минимума вместо глобального или вообще отсутствие найденных экстремальных точек.

Постановка задачи

Будет рассматриваться *непрерывная детерменированная задача многомерной безусловной оптимизации (минимизации функции)*:

$$\min_{X \in D \subset \mathbb{R}^{|X|}} f(X) = f^*, \quad X = (x_1, \dots, x_{|X|})^T,$$

где X — вектор варьируемых параметров, $|X|$ — его размерность;

f^* — искомое решение, то есть минимальное значение функции в рассматриваемой области поиска;

$D \subset \mathbb{R}^{|X|}$ — область поиска, заданная неравенствами:

$$D = \{X \in \mathbb{R}^{|X|} : x_i \geq x_i^-, x_i \leq x_i^+, i \in 1 : |X|\}$$

В общем случае рассмотренный далее алгоритм не предполагает задание каких-либо ограничений на область поиска D , однако в целях значительного уменьшения вычислительной сложности алгоритма такие ограничения будут наложены.

Алгоритм роя частиц

Предыстория

Канонический алгоритм был предложен Кеннеди и Эберхартом в 1995 году. Известно большое количество модификаций этого алгоритма, в которых изменены принципы коммуникации агентов популяции. В основу данного алгоритма легли поведенческие модели толпы людей, стаи птиц и прочие примеры живой природы. Особенностью таких моделей является цельное поведение при отсутствии центра управления за счет частичной коммуникации агентов между собой.

Далее будет рассмотрен канонический алгоритм роя частиц (*PSO*), а также его программная реализация и результаты тестов на различных функциях.

Важным замечанием также является тот факт, что в качестве **фитнес-функции**, то есть функции, используемой для вычисления положений частиц и для оценки текущего приближения к решению задачи, будет использована целевая функция, хоть и существует множество других способов определения фитнес-функции (например, в качестве фитнес-функции может быть использована обратная к целевой функция).

Описание канонического алгоритма

Алгоритм является алгоритмом стохастической глобальной безусловной оптимизации. Однако, как уже было отмечено выше, на область поиска будут наложены ограничения в виде неравенств, хоть общий вид алгоритма и не предполагает каких-либо условий.

Общая идея алгоритма состоит в том, что поиск минимума целевой функции осуществляет популяция агентов-частиц, именуемая роем. Далее рой частиц будет обозначаться как S (*swarm*), а количество агентов в популяции соответственно как $|S|$. Поиск глобального минимума осуществляется за счет перемещения частиц в пространстве поиска.

Каждую частицу s_i определяют несколько параметров:

- **Текущее положение в пространстве** — вектор $X_i = (x_{1,i}, \dots, x_{|X|,i}) \in \mathbb{R}^{|X|}$
- **Текущая скорость частицы** — вектор $V_i = (v_{1,i}, \dots, v_{|X|,i}) \in \mathbb{R}^{|X|}$
- **Значение фитнес-функции частицы** — значение функции в точке ее текущего положения, то есть $\varphi_i = \varphi(X_i)$
- **Лучшее значение фитнес-функции частицы** за все совершенные итерации алгоритма, а также положение, в котором это значение было достигнуто, то есть

$$X_i^p : \varphi(X_i^p) = \min_{t \in 1:\hat{t}} \{\varphi(X_i^t)\}.$$

Здесь t — номер итерации, \hat{t} — количество итераций, совершенных на данный момент.

Использован индекс p (*private*), так как вектор X_i^p в процессе итераций образует так называемый **собственный путь** (*private guide*)

- Множество индексов соседей частицы N (*neighbours*) : $|N| \leq |S| - 1$
- Лучшее значение фитнес-функции среди всех соседей на данной итерации, а также положение этого соседа, то есть

$$X_i^l : \varphi(X_i^l) = \min_{j \in N} \{\varphi(X_j)\}.$$

Здесь j — индекс частицы-соседа, имеющей наименьшее значение фитнес-функции на данной итерации.

Использован индекс l (*local*), так как вектор X_i^l в процессе итераций образует так называемый **локальный путь** (*local guide*)

Инициализация начальной популяции:

При создании начальной популяции положением каждой частицы принимается случайное положение в области поиска, а скоростью — случайный вектор, находящийся в пределах области поиска. При этом все частицы роя изначально равномерно распределены в области поиска D .

Вычисление нового положения частицы вычисляется по следующему правилу:

$$X_i^{t+1} = X_i(t+1) = X_i(t) + V_i(t+1) = X_i^t + V_i^{t+1}$$

$$V_i^{t+1} = V_i(t+1) = b_i \cdot V_i(t) + b_c \cdot rnd[0; 1] \cdot X_i^p(t) + b_s \cdot rnd[0; 1] \cdot X_i^l(t)$$

Здесь $rnd(0; 1)$ — псевдослучайное вещественное число из отрезка $[0; 1]$;

b_i — **инерционная** компонента, отвечающая направлению движения частицы на предыдущей итерации;

b_c — **когнитивная** компонента, отвечающая стремлению частицы вернуться к лучшему достигнутому значению фитнес-функции;

b_s — **социальная** компонента, отвечающая стремлению частицы приблизиться к лучшему значению фитнес-функции, достигнутому ее соседями на данной итерации.

Компоненты b_i, b_c, b_s являются *свободными параметрами* алгоритма, рекомендованными значениями которых являются

$$b_i = 0,7298; b_c = b_s = 1,49618$$

Именно эти значения будут использованы при реализации алгоритма в дальнейшем.

Условия завершения работы алгоритма:

В качестве условий завершения действия алгоритма будут использована комбинация двух условий:

- **Ограничение на количество итераций:**

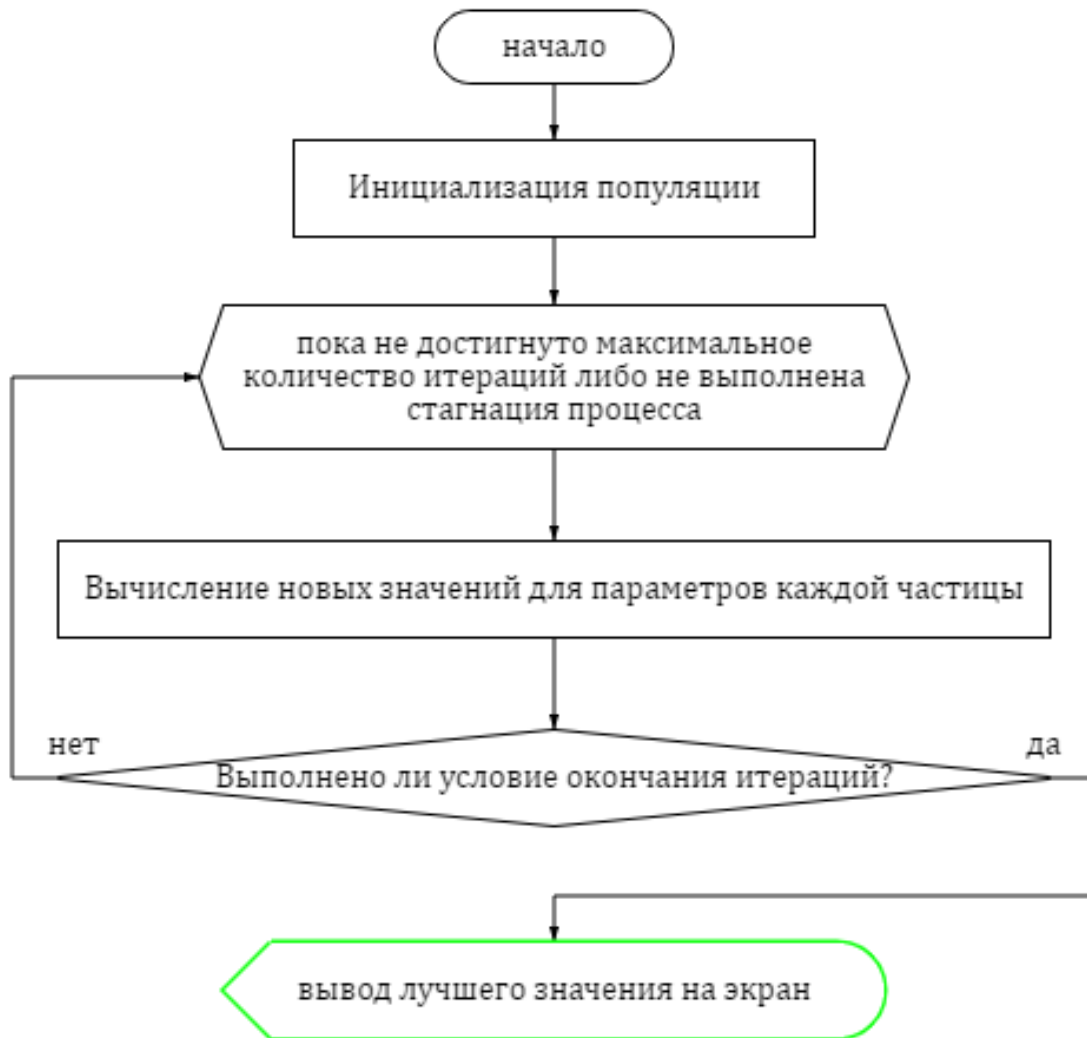
При достижении максимально допустимого количества итераций работа алгоритма приостанавливается, лучшее найденное решение принимается за ответ.

- **Стагнация процесса:**

Если в течение заданного количества итераций лучшее решение не изменяется, то наблюдается так называемая стагнация процесса. При достижении стагнации работа алгоритма приостанавливается, лучшее найденное решение принимается за ответ

Схема алгоритма в общем виде:

Общий вид алгоритма может быть представлен в виде следующей блок-схемы:



блок-схема, отражающая работу алгоритма

Заполнение множества индексов соседей. Топологии соседства частиц

Топология соседства — это правило, по которому для каждой частицы определяются ее частицы-соседи и заполняется множество их индексов.

Существуют **статические** и **динамические** топологии соседства частиц. Их отличие в том, что динамическая топология изменяет правило заполнения множества индексов соседей во время работы алгоритма, а статическая топология позволяет выполнить одно заполнение при инициализации роя частиц и не изменяет множества индексов частиц во время итераций алгоритма. Далее будут рассмотрены несколько статических топологий.

Топология соседства может быть однозначно представлена в виде связного неориентированного **графа соседства**, в котором *вершинами* являются частицы роя, а *ребрами* соединяются те вершины-частицы, которые являются соседями. При этом соседство частиц не зависит от их геометрического положения в пространстве поиска, а определяется исключительно индексом частицы во всем рое. Это позволяет упростить вычисления множества индексов соседей, а также способствует диверсификации поиска, то есть его расширению.

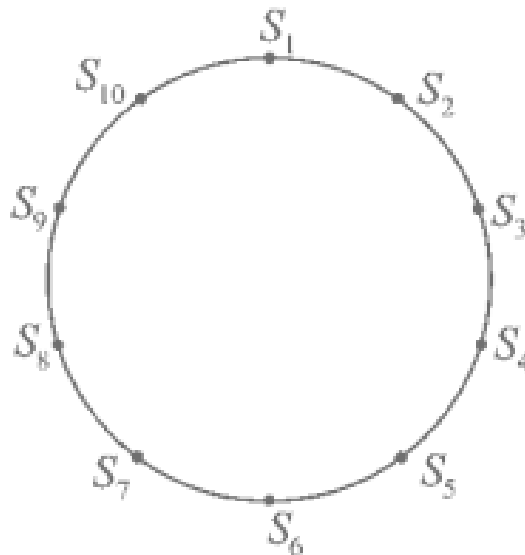
Важным показателем графа соседства является его **диаметр**, то есть наименьшее число ребер, по которому можно попасть из одной вершины в другую. Диаметр графа соседства показывает, насколько быстро распространяется информация среди частиц о положениях их соседей.

В данной работе будет рассмотрено 4 топологии соседства частиц:

- Топология “кольцо” (*ring*)
- Топология “клика” (*clique*)
- Топология “двумерный тор” (*torus*)
- Топология “кластер” (*cluster*)

Топология “кольцо”

В данной топологии граф соседства частиц имеет вид кольца:



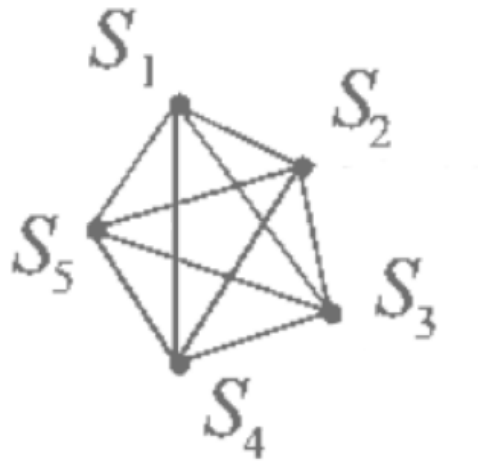
топология “кольцо” для $|S| = 10$

В данном случае каждая частица имеет ровно 2 соседа. При этом диаметр графа равен $\frac{|S|}{2}$, так как для того, чтобы достичь противоположной на окружности точки необходимо пройти $\frac{|S|}{2}$ ребер для четного $|S|$ и $\frac{|S| - 1}{2}$ для нечетного $|S|$

Поскольку диаметр графа достаточно велик, информация о положениях соседей распространяется медленно и такая топология гарантирует высокую диверсификацию алгоритма, но низкую его интенсивность. Поэтому такую топологию целесообразно использовать для многоэкстремальных функций, чтобы повысить вероятность нахождения глобального решения.

Топология “клика”

В данной топологии граф соседства частиц соответствует названию и является кликой:



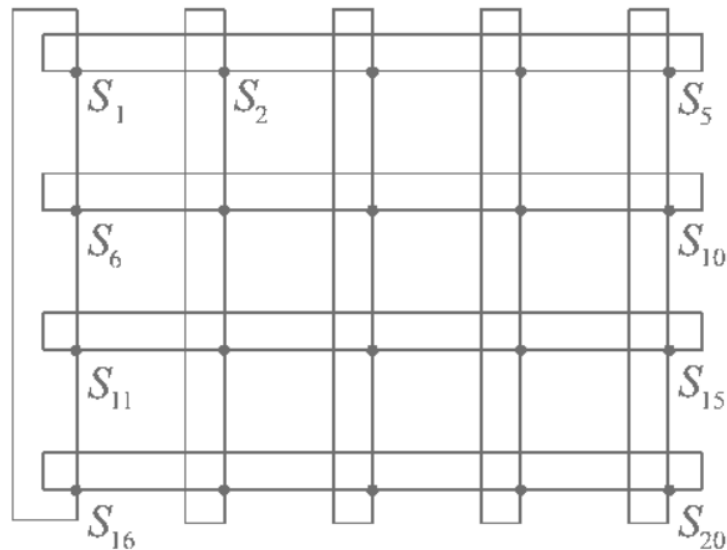
топология “клика” для $|S| = 5$

В данном случае каждая частица имеет $|S| - 1$ соседей. При этом диаметр графа равен 1, так как все вершины соединены напрямую между собой по определению клики.

Поскольку диаметр графа очень мал, информация о положениях соседей распространяется очень быстро и такая топология гарантирует высокую интенсивность алгоритма, но в то же время низкую диверсификацию поиска. Поэтому такую топологию целесообразно использовать для простых функций, которые при малой ширине поиска все еще позволяют определить глобальный минимум.

Топология “двумерный тор”

В данной топологии граф соседства частиц имеет непримитивный вид. Для топологии двумерного тора необходимо определить размеры “таблицы”, в каждой “ячейке” которой будет находиться частица. Таким образом для этой топологии количество агентов должно быть составным числом и, желательно, раскладываться в произведение достаточно близких чисел (размеров “таблицы”), чтобы сбалансировать интенсивность и ширину поиска. Частицы записываются в таблицу слева направо, сверху вниз, затем таблица сворачивается в горизонтальную “трубочку” и замыкается в тор:



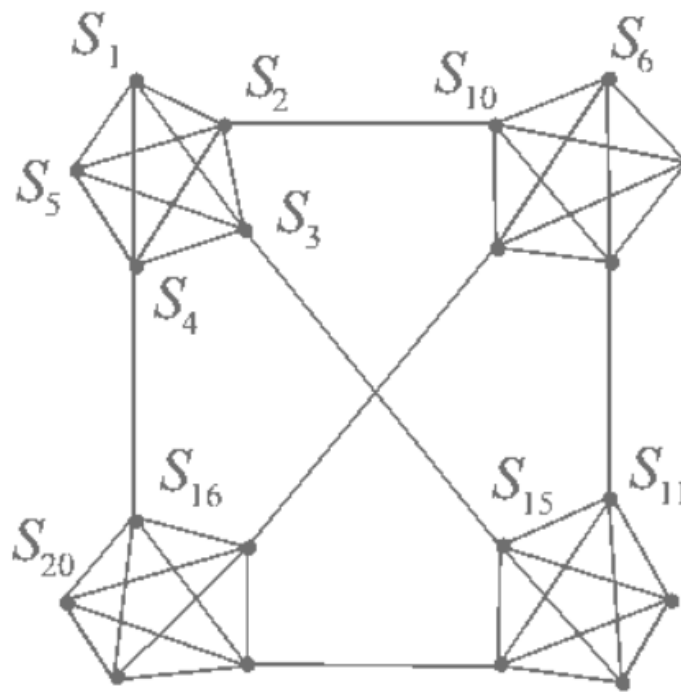
топология “двумерный тор” для $|S| = 20$

В данном случае каждая вершина имеет 4 соседа. Диаметр графа для таблицы размеров

$n \times m$ равен $\left\lfloor \frac{n}{2} \right\rfloor + \left\lfloor \frac{m}{2} \right\rfloor$. Эта величина лежит между диаметрами двух предыдущих топологий, и потому имеет средние показатели интенсивности поиска и его ширины. Именно поэтому данная топология может быть использована для большинства задач оптимизации.

Топология “кластер”

Данная топология является неким обобщением топологии “клика”, позволяющим снизить интенсивность поиска и увеличить диверсификацию. Граф соседства частиц имеет вид нескольких клик, соединенных между собой. При этом клики имеют одинаковые размеры, то есть количество клик должно делить общее количество частиц. Более того, поскольку степень каждой вершины графа не может превышать количество вершин в клике, то количество клик не может превышать количество вершин в клике больше, чем на 1, так как в противном случае не выйдет соединить эту клику с остальными так, чтобы из каждой вершины выходило только по одному “внешнему” ребру:



топология “кластер” для $|S| = 20$, количество клик — 4

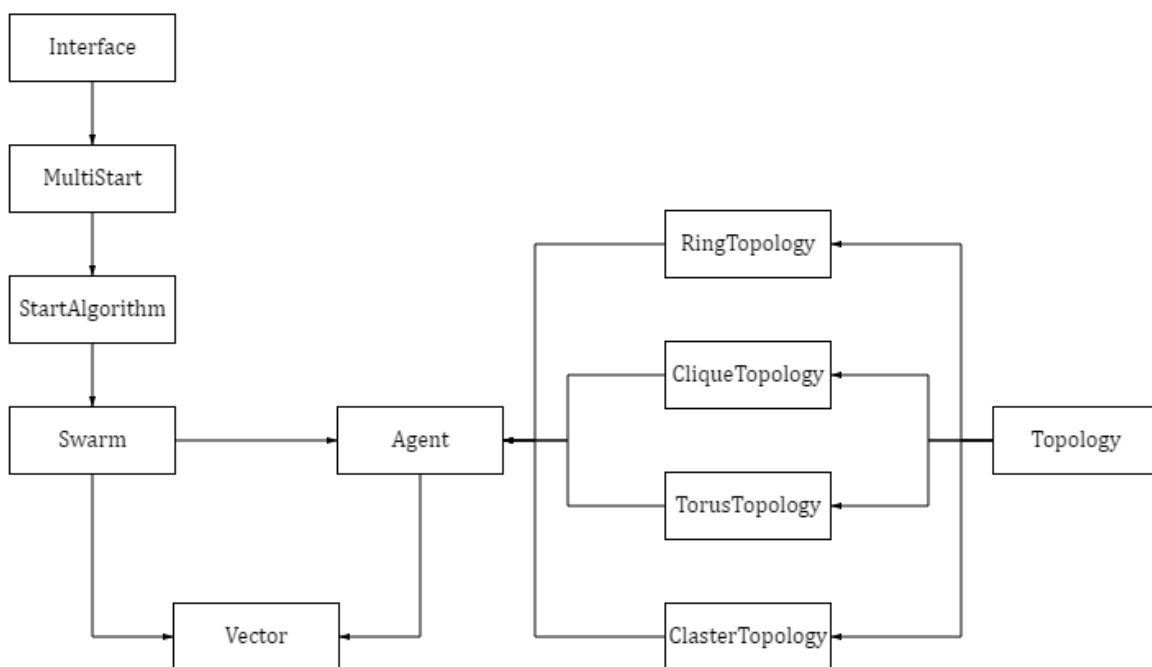
Диаметр такого графа равен 3, так как худший случай получается, например, для вершин S_1 и S_6 . Эта величина при достаточно больших размерах популяции меньше диаметра графа для топологии двумерного тора, однако она все еще превосходит диаметр графа для клики и будет давать несколько большую широту поиска и несколько более медленный темп схождения алгоритма. Такая топология также подойдет для большинства задач оптимизации.

Программная реализация алгоритма

Алгоритм был реализован на языке программирования Java, который предполагает объектно-ориентированный подход. Поскольку в алгоритме достаточно четко прослеживается иерархичная структура и уровни абстракции, он хорошо подходит для реализации на объекто-ориентированном языке программирования.

Также важно отметить, что в большинстве популяционных алгоритмов используется метод мультистарта — нескольких запусков программы и затем выбора лучшего ответа из полученных. Этот подход позволяет уменьшить влияние случайности перемещения агентов и их изначальной инициализации на итоговый ответ.

Структура программы имеет следующий вид:



Использованы следующие классы:

- ***Vector.java*** — класс, описывающий структуру вектора и стандартные операции с векторами, такие как сложение и умножение на скаляр
- ***Agent.java*** — класс, описывающий структуру агента поиска (частицы)
- ***Swarm.java*** — класс, описывающий структуру всего роя частиц и его поведения
- ***StartAlgorithm.java*** — класс, позволяющий запустить алгоритм и получить минимальное значение оптимизируемой функции
- ***MultiStart.java*** — класс, запускающий алгоритм некоторое введенное пользователем количество раз. При этом запущенные алгоритмы выполняются параллельно
- ***Interface.java*** — класс, содержащий все необходимое для запуска всей программы, в том числе и метод ***main***
- ***FreeParameters.java*** — класс, хранящий все свободные параметры алгоритма. Параметры вводятся пользователем один раз перед началом работы алгоритма
- ***Topology.java*** — интерфейс, реализуемый классами различных топологий и содержащий все необходимое для этой реализации
- ***RingTopology.java, CliqueTopology.java, TorusTopology.java, ClasterTopology.java*** — классы, реализующие интерфейс ***Topology*** и содержащие методы, которые заполняют массив индексов соседей каждой частицы в соответствии с выбранной топологией

Класс *FreeParameters*

Поскольку алгоритм содержит большое количество свободных параметров, которые не изменяются в течение его работы и даже при повторных запусках (метод **мультистарта**), было принято решение создать класс, который хранит все эти параметры и один раз за запуск программы просит пользователя ввести их значения. Также этот класс позволяет абстрагироваться от конкретных параметров, которые необходимо передавать тем или иным методам других классов, что хоть и не сильно, но отрицательно влияет на память программы, однако упрощает понимание.

```
public class FreeParameters {
    public int agentsCount;
    public int dimension;
    public double[] minimumRestrictions;
    public double[] maximumRestrictions;
    public double inertialComponent;
    public double cognitiveComponent;
    public double socialComponent;
    public int maximumIterationsNumber;
    public int stagnationLimit;
    public int multiStartNumber;
    public int topologyType;
    public int torusSize;
    public int cliquesCount;
```

Класс содержит в себе достаточно много полей:

- ***agentsCount*** — количество частиц-агентов в пространстве поиска
- ***dimension*** — размерность пространства поиска
- ***minimumRestrictions*** — нижние ограничения на каждую координату области поиска, то есть массив x_i^- в обозначениях постановки задачи
- ***maximumRestrictions*** — верхние ограничения на каждую координату области поиска, то есть массив x_i^+ в обозначениях постановки задачи
- ***inertialComponent*** — инерционная составляющая, рекомендуемое значение — 0,7298
- ***cognitiveComponent*** — когнитивная составляющая, рекомендуемое значение — 1,49618

- ***socialComponent*** — социальная составляющая, рекомендуемое значение — 1,49618
- ***maximumIterationsNumber*** — максимальное количество итераций, при превышении которого алгоритм заканчивает свою работу
- ***stagnationLimit*** — количество итераций, необходимое для завершения алгоритма из-за стагнации — если на протяжении этого количества итераций лучшее решение алгоритма не изменялось, то алгоритм заканчивает свою работу
- ***multiStartNumber*** — количество запусков алгоритма
- ***topologyType*** — тип топологии для заполнения массивов индексов соседей у каждой частицы
- ***torusSize*** — при выборе топологии двумерного тора пользователь должен ввести количество столбцов в “таблице”
- ***cliquesCount*** — при выборе топологии кластера пользователь должен ввести количество клик в кластере

```
public FreeParameters() throws NoSuchElementException {
    Scanner scanner = new Scanner(System.in);
    scanner.useLocale(Locale.US);
    System.out.print("Insert the number of agents - particles in the search area\n"
        + "Number of agents: ");
    this.agentsCount = scanner.nextInt();
    System.out.print("Insert the dimension of the search area\n" + "Dimension: ");
    this.dimension = scanner.nextInt();
    System.out.print("Insert minimum restrictions for each coordinate of" +
        " particle position\n");
    this.minimumRestrictions = new double[this.dimension];
    for(int i = 0; i < this.dimension; i++){
        System.out.print("Minimum restriction for " + (i+1) + " coordinate: ");
        this.minimumRestrictions[i] = scanner.nextDouble();
    }
    System.out.print("Insert maximum restrictions for each coordinate of" +
        " particle position\n");
    this.maximumRestrictions = new double[this.dimension];
    for(int i = 0; i < this.dimension; i++){
        System.out.print("Maximum restriction for " + (i+1) + " coordinate: ");
        this.maximumRestrictions[i] = scanner.nextDouble();
    }
    System.out.print("Insert inertial component\n" + "Inertial component: ");
    this.inertialComponent = scanner.nextDouble();
    System.out.print("Insert cognitive component\n" + "Cognitive component: ");
}
```

```

this.cognitiveComponent = scanner.nextDouble();
System.out.print("Insert social component\n" + "Social component: ");
this.socialComponent = scanner.nextDouble();

```

Конструктор *FreeParameters()* запрашивает у пользователя значения всех свободных параметров через консоль. После ввода значения размерности области поиска выделяется память под массивы ограничений. Считывание значений производится с помощью класса *java.util.Scanner*.

```

System.out.print("Insert the number of the topology you wish to select:\n" +
    "1. Ring topology\n2. Clique topology\n3. Torus topology\n" +
    "4. Cluster topology\n");
this.topologyType = scanner.nextInt();
if(this.topologyType == Topology.TORUS){
    System.out.print("For this topology you have to insert the size of" +
        " the torus\n");
    this.torusSize = scanner.nextInt();
    while(this.agentsCount % this.torusSize != 0) {
        System.out.print("Agents amount = " + this.agentsCount +
            " cannot be divided by the size of torus = "
            + this.torusSize + "\n");
        this.torusSize = scanner.nextInt();
    }
}
if(this.topologyType == Topology.CLUSTER){
    System.out.print("For this topology you have to insert the amount of" +
        " cliques in the neighbourhood graph\n");
    this.cliquesCount = scanner.nextInt();
    while(this.agentsCount % this.cliquesCount != 0 || this.agentsCount /
        this.cliquesCount < this.cliquesCount - 1) {
        System.out.print("Agents amount = " + this.agentsCount +
            " cannot be divided by the number of cliques = "
            + this.cliquesCount + "\n");
        this.cliquesCount = scanner.nextInt();
    }
}
System.out.print("Insert maximum number of algorithm iterations\n" +
    "Maximum number of iterations: ");
this.maximumIterationsNumber = scanner.nextInt();
System.out.print("Insert needed number of iterations while stagnation" +
    " of the process\n" + "Stagnation limit: ");
this.stagnationLimit = scanner.nextInt();
System.out.print("Insert number of algorithm starts\n" +
    "Multi start executions number: ");
this.multiStartNumber = scanner.nextInt();
}
}

```

После ввода типа топологии производится проверка, и если тип топологии это двумерный тор либо кластер, то метод запросит необходимые для этих топологий параметры, причем считывание будет производиться до тех пор, пока они не начнут удовлетворять требованиям, описанным в разделе о топологиях соседства. Затем считываются оставшиеся свободные параметры и работа конструктора заканчивается.

Класс *Vector*

Язык Java имеет встроенные пакеты с уже готовыми классами для векторов, однако было принято решение написать такой класс самостоятельно. Класс содержит всю информацию о векторе и такие операции с векторами как сложение, умножение на скаляр.

```
public class Vector {
    public int dimension;
    public double[] coordinates;

    public Vector(int dimension){
        this.dimension = dimension;
        this.coordinates = new double[dimension];
    }
    public Vector(double[] coordinates){
        this.coordinates = coordinates;
        this.dimension = coordinates.length;
    }

    public Vector sum(Vector operand) throws Exception{
        if(this.dimension != operand.dimension)
            throw new Exception("Vectors cannot be added: difference in operands'" +
                                " dimensions\n");
        double[] coordinatesSum = new double[this.dimension];
        for(int i = 0; i < this.dimension; i++)
            coordinatesSum[i] = this.coordinates[i] + operand.coordinates[i];
        return new Vector(coordinatesSum);
    }
    public Vector scalarMultiplication(double scalar){
        double[] newCoordinates = new double[this.dimension];
        for(int i = 0; i < this.dimension; i++)
            newCoordinates[i] = this.coordinates[i] * scalar;
        return new Vector(newCoordinates);
    }
}
```

Класс содержит всего два поля:

- ***dimension*** — размерность векторного пространства
- ***coordinates*** — массив координат вектора

В классе есть два конструктора:

- ***Vector(int dimension)*** — создает нулевой вектор заданной размерности
- ***Vector(double[] coordinates)*** — создает вектор по полученному массиву координат. Размерность при этом определяется длиной массива

Наконец, реализованы два оператора:

- ***Vector sum(Vector operand)*** — метод, суммирующий два вектора по координатам и возвращающий результат сложения
- ***Vector scalarMultiplication(double scalar)*** — метод, умножающий вектор по координатам на полученный скаляр и возвращающий результат умножения

Класс *Agent*

Этот класс содержит в себе всю информацию о частице, а также методы ее перемещения и обновления параметров, подробно описанных в разделе о работе алгоритма.

```
public class Agent {
    public Vector currentPosition;
    public Vector velocity;
    public ArrayList<Integer> neighbours;
    public Vector bestFitnessFunctionArgument;
    public double bestFitnessFunctionValue;
    public Vector bestNeighbourPosition;
    public double bestNeighbourFitnessFunctionValue;
    public int index;

    public Agent(int index, FreeParameters freeParameters) {
        this.index = index;
        Random random = new Random(System.currentTimeMillis());
        this.currentPosition = new Vector(freeParameters.dimension);
        this.velocity = new Vector(freeParameters.dimension);
        for (int i = 0; i < freeParameters.dimension; i++) {
            double residue = freeParameters.maximumRestrictions[i] -
                freeParameters.minimumRestrictions[i];
            this.currentPosition.coordinates[i] = random.nextDouble() *
                residue + freeParameters.minimumRestrictions[i];
            this.velocity.coordinates[i] = random.nextDouble() * 2 * residue - residue;
        }
    }
}
```

```

        this.bestFitnessFunctionArgument = this.currentPosition;
        this.bestFitnessFunctionValue = getFitnessFunctionValue(this.currentPosition);
        this.bestNeighbourPosition = this.currentPosition;
        this.bestNeighbourFitnessFunctionValue = Double.MAX_VALUE;
    }

```

Класс содержит поля, определяющие параметры частицы:

- ***currentPosition*** — вектор, определяющий позицию частицы в области поиска в данный момент
- ***velocity*** — вектор, определяющий скорость частицы в данный момент
- ***neighbours*** — связный список целых чисел, содержащий индексы всех соседей данной частицы. Использован список вместо массива, так как Java не поддерживает изменение длины массива после его инициализации, а количество соседей не всегда заранее известно
- ***bestFitnessFunctionArgument*** — вектор позиции данной частицы, в которой она имела лучшее значение фитнес-функции по всем итерациям до данной. Именно этот вектор образует приватный путь частицы
- ***bestFitnessFunctionValue*** — значение фитнес-функции в лучшей точке, хранящейся в поле, описанном выше
- ***bestNeighbourPosition*** — вектор позиции соседа данной частицы, который имеет лучшее значение фитнес-функции на данной итерации. Именно этот вектор образует локальный путь частицы
- ***bestNeighbourFitnessFunctionValue*** — значение фитнес-функции в лучшей точке среди соседей на данной итерации
- ***index*** — индекс данной частицы в общем роле. Имеет значение от 0 до $|S| - 1$

Конструктор *Agent(int index, FreeParameters freeParameters)* присваивает значение индекса соответствующему полю частицы. Далее создается объект класса ***java.util.Random***, выполняющий роль генератора псевдослучайных чисел. После выделения памяти под векторы позиции и скорости их координаты заполняются случайными числами, находящимися в пределах области поиска. Особое удобство ***Random.nextDouble()*** в том, что эта функция возвращает псевдослучайное число в промежутке $[0, 1]$. Наконец, лучшему приватному положению этой частицы присваивается начальная позиция, лучшему положению

соседа, в силу отсутствия информации о них, также начальная позиция. При этом в качестве значения фитнес-функции соседа устанавливается константа ***Double.MAX_VALUE***, и при сравнении на следующих итерациях реальных значений фитнес-функции с максимально возможным вещественным числом поле будет изменено.

```
public double getFitnessFunctionValue(Vector x) {
    double result = 0;
    //insert function calculation rule here
    return result;
}
```

Этот метод предназначен для вычисления значения фитнес-функции для конкретного вектора. В силу обобщенности оптимизируемых функций нет никакой возможности каким-либо образом считать их из консоли при вводе пользователя, поэтому реализация данного метода уникальна для каждой новой задачи оптимизации и выполняется пользователем.

```
public void setNeighbours(FreeParameters freeParameters) throws Exception {
    this.neighbours = new ArrayList<>();
    switch (freeParameters.topologyType) {
        case Topology.RING:
            RingTopology ringTopology = new RingTopology();
            ringTopology.agentNeighbourhood(this, freeParameters.agentsCount);
            break;
        case Topology.CLIQUE:
            CliqueTopology cliqueTopology = new CliqueTopology();
            cliqueTopology.agentNeighbourhood(this, freeParameters.agentsCount);
            break;
        case Topology.TORUS:
            TorusTopology torusTopology = new TorusTopology(freeParameters.torusSize);
            torusTopology.agentNeighbourhood(this, freeParameters.agentsCount);
            break;
        case Topology.CLUSTER:
            ClusterTopology clusterTopology = new ClusterTopology
                (freeParameters.agentsCount, freeParameters.cliquesCount);
            clusterTopology.agentNeighbourhood(this, freeParameters.agentsCount);
            break;
        default:
            throw new Exception("Topology with such a number does not exist\n");
    }
}
```

Данный метод необходим для заполнения списка индексов соседей частицы. После выделения памяти происходит проверка выбранного типа топологии и в зависимости от выбора пользователя вызываются соответствующие методы, описанные в классах различных топологий. Реализация этих методов будет рассмотрена позже.

```
public void nextPosition(FreeParameters freeParameters) throws Exception {
    Random random = new Random(System.currentTimeMillis());
    Vector newVelocity;
    Vector inertia = this.velocity.scalarMultiplication(freeParameters.
        inertialComponent);
    Vector cognition = this.currentPosition.scalarMultiplication(-1).
        sum(this.bestFitnessFunctionArgument).scalarMultiplication
        (freeParameters.cognitiveComponent).scalarMultiplication(random.nextDouble());
    Vector socialisation = this.currentPosition.scalarMultiplication(-1).
        sum(this.bestNeighbourPosition).scalarMultiplication
        (freeParameters.socialComponent).scalarMultiplication(random.nextDouble());
    newVelocity = inertia.sum(cognition).sum(socialisation);

    Vector newPosition;
    newPosition = this.currentPosition.sum(newVelocity);
    for (int i = 0; i < newVelocity.dimension; i++) {
        if (newPosition.coordinates[i] > freeParameters.maximumRestrictions[i])
            newPosition.coordinates[i] = freeParameters.maximumRestrictions[i] -
                random.nextDouble() * (freeParameters.maximumRestrictions[i] -
                    freeParameters.minimumRestrictions[i]);
        if (newPosition.coordinates[i] < freeParameters.minimumRestrictions[i])
            newPosition.coordinates[i] = freeParameters.minimumRestrictions[i] +
                random.nextDouble() * (freeParameters.maximumRestrictions[i] -
                    freeParameters.minimumRestrictions[i]);
    }
    this.velocity = newVelocity;
    this.currentPosition = newPosition;
}
```

Этот метод предназначен для вычисления новых векторов положения и скорости для данной частицы. Снова используется генератор псевдослучайных чисел, а затем происходит вычисление нового вектора скорости по правилу, описанному в разделе описания алгоритма. Новый вектор скорости является суммой трех компонент - инерциальной, когнитивной и социальной. Новый вектор положения получается прибавлением к предыдущему положению вектора новой скорости. После происходит проверка на принадлежность новой точки области поиска и в случае, если позиция частицы вышла за границы, ей присваивается новая позиция внутри области. После этого происходит переписывание полей новых векторов.


```

public void privateGuideNextPosition() {
    double newFitnessFunctionValue = getFitnessFunctionValue(currentPosition);
    if(newFitnessFunctionValue < bestFitnessFunctionValue){
        this.bestFitnessFunctionValue = newFitnessFunctionValue;
        this.bestFitnessFunctionArgument = currentPosition;
    }
}
}

```

Данный метод используется для вычисления новой позиции приватного пути — иначе говоря, новые значения полей лучшего значения фитнес-функции данной точки и аргумента этого значения. Если в текущей позиции значение функции оказалось лучше (то есть меньше) известного лучшего значения, то происходит переприсваивание.

На этом реализация класса **Agent** заканчивается.

Класс *Swarm*

Класс, отвечающий модели роя частиц, соответственно, содержащий массив этих частиц, а также метод, который фактически является одной итерацией алгоритма.

```

public class Swarm {
    public Agent[] particles;
    public int agentsCount;

    public Swarm(FreeParameters freeParameters) throws Exception {
        this.agentsCount = freeParameters.agentsCount;
        this.particles = new Agent[agentsCount];
        for(int i = 0; i < agentsCount; i++){
            this.particles[i] = new Agent(i, freeParameters);
            this.particles[i].setNeighbours(freeParameters);
        }
    }
}

```

Класс содержит всего два поля:

- **particles** — массив частиц
- **agentsCount** — количество частиц, перемещающихся в области поиска

Конструктор *Swarm(FreeParameters freeParameters)* присваивает полю значение количества агентов, введенное пользователем. Затем выделяется

память под массив частиц и каждая частица отдельно инициализируется с уникальным индексом во всем рое. Наконец, у каждой частицы с помощью описанного выше метода заполняется список индексов соседей.

```
public void swarmLocalGuideNextPosition() {
    for(int i = 0; i < this.agentsCount; i++){
        for(int j = 0; j < this.particles[i].neighbours.size(); j++){
            Agent neighbour = this.particles[this.particles[i].neighbours.get(j)];
            if(neighbour.getFitnessFunctionValue(neighbour.currentPosition) <
                this.particles[i].bestNeighbourFitnessFunctionValue){
                this.particles[i].bestNeighbourFitnessFunctionValue =
                    neighbour.getFitnessFunctionValue(neighbour.currentPosition);
                this.particles[i].bestNeighbourPosition = neighbour.currentPosition;
            }
        }
    }
}
```

Данный метод также необходим для обновления полей класса **Agent** на каждой итерации, однако в силу того, что он задействует другие частицы, метод был реализован в классе **Swarm**.

У каждой частицы в рое рассматривается каждый ее сосед. Если на данной итерации значение фитнес-функции у соседа лучше, чем имеющееся лучшее значение в соответствующем поле частицы, соседи которой рассматриваются, то происходит переприсваивание.

```
public void nextIteration(FreeParameters freeParameters) throws Exception{
    for(int i = 0; i < this.agentsCount; i++){
        particles[i].nextPosition(freeParameters);
        particles[i].privateGuideNextPosition();
    }
    this.swarmLocalGuideNextPosition();
}
}
```

Метод, соответствующей одной итерации алгоритма и изменяющий все динамические поля каждой частицы. У каждой частицы обновляется ее положение, приватный и локальный пути.

На этом реализация класса **Swarm** заканчивается.

Интерфейс *Topology*

Интерфейс создан для обобщения различных видов топологий и хранения константных переменных.

```
public interface Topology {
    int RING = 1;
    int CLIQUE = 2;
    int TORUS = 3;
    int CLASTER = 4;
    void agentNeighbourhood(Agent particle, int agentsCount);
}
```

Интерфейс содержит 4 поля, определяющих различные виды топологий. Таким образом при вводе вида топологии пользователь вводит ее номер, соответствующий значению константного поля данного интерфейса, но для повышения уровня абстракции, например, в методе ***Agent.setNeighbours()*** при проверке типа топологии указаны не конкретные числа, а константы вида ***Topology.RING***, ***Topology.CLIQUE*** и т.д.

Также интерфейс содержит определение метода для заполнения списка соседей частицы, и в силу особенностей языка этот метод должен быть переопределен в каждом классе топологии, реализующей данный интерфейс.

Класс *RingTopology*

Поскольку данная топология не требует никаких дополнительных данных, класс содержит исключительно переопределение метода ***agentNeighbourhood***.

```
public class RingTopology implements Topology {
    @Override
    public void agentNeighbourhood(Agent particle, int agentsCount){
        if(particle.index == 0)
            particle.neighbours.add(agentsCount - 1);
        else
            particle.neighbours.add(particle.index - 1);
        if(particle.index == agentsCount - 1)
            particle.neighbours.add(0);
        else
            particle.neighbours.add(particle.index + 1);
    }
}
```

Частицы объединяются в кольцо по порядку, то есть для частицы с индексом i соседями станут $i - 1, i + 1$. Исключение составляют частицы с индексом 0 и $|S| - 1$, поэтому они рассмотрены отдельно.

Класс *CliqueTopology*

Эта топология также не предполагает никаких дополнительных данных, и класс по аналогии с топологией кольца содержит только переопределенный метод ***agentNeighbourhood***.

```
public class CliqueTopology implements Topology {
    @Override public void agentNeighbourhood(Agent particle, int agentsCount){
        for(int i = 0; i < agentsCount; i++){
            if(particle.index != i)
                particle.neighbours.add(i);
        }
    }
}
```

Поскольку в клике в графе соседства частица соединена ребрами со всеми остальными частицами в рое, в список соседей добавляются все индексы кроме индекса самой частицы.

Класс *TorusTopology*

Данная топология требует ввода пользователем длины “таблицы”, а потому класс содержит поле, хранящее эту величину, конструктор и переопределенный метод.

```
public class TorusTopology implements Topology {
    public int torusSize;

    public TorusTopology(int torusSize){
        this.torusSize = torusSize;
    }

    @Override
    public void agentNeighbourhood(Agent particle, int agentsCount) {
        int lowerRounding = particle.index - particle.index % torusSize;
        particle.neighbours.add((torusSize + particle.index - 1) % torusSize +
            lowerRounding); // "left" neighbour
        particle.neighbours.add((torusSize + particle.index + 1) % torusSize +
            lowerRounding); // "right" neighbour
        particle.neighbours.add((particle.index + torusSize) % agentsCount);
    }
}
```

```

        // "lower" neighbour
        particle.neighbours.add((particle.index + agentsCount - torusSize) %
            agentsCount); // "upper" neighbour
    }
}

```

Поскольку определение метода ***agentNeighbourhood*** не может принимать на вход параметр ***torusSize***, то он хранится в данном классе как отдельное поле. В самом методе список соседей заполняется по порядку: левый, правый, нижний и верхний сосед частицы в торе.

Класс *ClusterTopology*

Данная топология также требует ввода пользователем вспомогательных данных, а именно — количества клик, поэтому ее класс аналогично предыдущей топологии содержит одно поле, одно вспомогательное поле, упрощающее расчеты, конструктор и переопределенный метод.

```

public class ClusterTopology implements Topology {
    public int cliquesCount;
    public int numberOfCliqueVertexes;

    public ClusterTopology(int agentsCount, int cliquesCount){
        this.cliquesCount = cliquesCount;
        this.numberOfCliqueVertexes = agentsCount / cliquesCount;
    }
    @Override
    public void agentNeighbourhood(Agent particle, int agentsCount){
        for(int i = particle.index - particle.index % this.numberOfCliqueVertexes;
            i < particle.index - particle.index % this.numberOfCliqueVertexes +
                this.numberOfCliqueVertexes; i++) //adding neighbours in the clique
        {
            if(i != particle.index)
                particle.neighbours.add(i);
        }
        int thisCliqueNumber = particle.index / this.cliquesCount + 1;
        int vertexNumber = particle.index % this.numberOfCliqueVertexes + 1;
        if((vertexNumber <= this.cliquesCount) && (vertexNumber != thisCliqueNumber)){
            int neighbourIndex = (vertexNumber - 1) * this.numberOfCliqueVertexes +
                thisCliqueNumber - 1;
            particle.neighbours.add(neighbourIndex);
        }
    }
}

```

В методе сначала в список добавляются все соседи из клики, в которой состоит данная частица, а затем происходит соединение клик между собой. При этом частице присваивается ее номер в клике (нумерация идет с 1 до ***numberOfCliqueVertexes***, при этом порядок нумерации соответствует общей индексации частиц — чем большее ее общий индекс, тем больше ее номер в клике), и каждая клика также нумеруется. Если номер частицы (например, j) в клике (например, с номером i) совпадает с номером существующей клики и эта клика — не та же, в которой содержится частица (то есть $i \neq j$), то она соединяется с этой кликой. При этом в клике, с которой происходит соединение (имеющей номер j), вершина выбирается аналогично — соседом станет та вершина, номер которой в клике j соответствует номеру клики, в которой состоит рассматриваемая частица (то есть i).

Класс *StartAlgorithm*

Класс, отвечающий за работу алгоритма в целом. Реализует интерфейс ***Runnable*** в целях дальнейшего распараллеливания работы нескольких алгоритмов.

Поскольку при переопределении метода ***run()*** в него не могут быть переданы никакие параметры, все необходимое для работы алгоритма хранится в его полях.

```
public class StartAlgorithm implements Runnable{
    public FreeParameters freeParameters;
    public double overallBestFitnessFunctionValue;
    private int stagnationCounter;
    public int iterationCounter;

    public StartAlgorithm(FreeParameters freeParameters){
        this.freeParameters = freeParameters;
        this.overallBestFitnessFunctionValue = Double.MAX_VALUE;
        this.stagnationCounter = 0;
        this.iterationCounter = 0;
    }
}
```

Здесь в поле ***freeParameters*** передается объект, содержащий все свободные параметры алгоритма, а остальные поля имеют более содержательное значение:

- ***overallBestFitnessFunctionValue*** — переменная, которая хранит лучшее среди всех частиц найденное значение фитнес-функции на какой-либо из прошедших итераций. Это же поле по окончании работы алгоритма содержит в себе найденное лучшее решение, то есть ответ

- **stagnationCounter** — счетчик, используемый для отслеживания стагнации алгоритма
- **iterationCounter** — счетчик итераций алгоритма, используется затем для вывода среднего количества итераций при нескольких вызовах алгоритма (мультистарте)

Конструктор **StartAlgorithm(FreeParameters freeParameters)** присваивает полю свободных параметров полученный объект. В поле лучшего решения записывается максимальное вещественное число — константа **Double.MAX_VALUE**, которая при сравнении с любым найденным решением на первой же итерации будет изменена на это решение. Счетчики стагнации и итераций полагаются равными нулю.

```
@Override
public void run(){
    try {
        Swarm swarm = new Swarm(this.freeParameters);
        double[] allFitnessFunctionValues =
            new double[this.freeParameters.agentsCount];
        for(this.iterationCounter = 0;
            this.iterationCounter < freeParameters.maximumIterationsNumber;
            this.iterationCounter++){
            if(this.stagnationCounter >= this.freeParameters.stagnationLimit) {
                return;
            }
            swarm.nextIteration(this.freeParameters);
            for(int i = 0; i < this.freeParameters.agentsCount; i++)
                allFitnessFunctionValues[i] =
                    swarm.particles[i].getFitnessFunctionValue(
                        swarm.particles[i].currentPosition);
            Arrays.sort(allFitnessFunctionValues);
            double bestFitnessFunctionValue = allFitnessFunctionValues[0];
            if(this.overallBestFitnessFunctionValue > bestFitnessFunctionValue){
                this.overallBestFitnessFunctionValue = bestFitnessFunctionValue;
                this.stagnationCounter = 0;
            }
            else
                this.stagnationCounter++;
        }
    }
    catch (Exception exception) {
        exception.printStackTrace();
    }
}
```

Метод ***run()*** представляет собой всю работу одного прогона алгоритма. Сначала инициализируется рой частиц, затем выделяется память под массив, в который будут записываться полученные на данной итерации значения фитнес-функций каждой частицы.

Далее в цикле, реализующем ограничение на количество итераций, производится проверка счетчика стагнации и если он превысил минимальный лимит, то алгоритм завершает свою работу. Если же этого не произошло, то всем частицам присваиваются новые значения для позиций и прочих динамических полей, так как происходит вызов метода ***Swarm.nextIteration***.

После того, как все частицы изменили свое положение в пространстве поиска, происходит запись в соответствующий массив всех значений фитнес-функций, полученных на данной итерации. Массив затем сортируется и лучшим по всем частицам значением фитнес-функции на данной итерации принимается его первый элемент, так как он минимален. После, если найденное решение на данной итерации лучше, чем уже имеющееся решение, происходит переприсваивание, а счетчик стагнации принимает нулевое значение, так как минимальное значение фитнес-функции изменилось. Если же этого не произошло, то решение осталось прежним, и потому счетчик стагнации повышается на 1.

Вне зависимости от причины завершения работы алгоритма лучшее значение будет храниться в поле ***overallBestFitnessFunctionValue***.

Класс ***MultiStart***

Данный класс предназначен для запуска алгоритма некоторое количество раз, и в целях ускорения вычислительных процессов алгоритм запускается параллельно.

```
public class MultiStart{
    public FreeParameters freeParameters;
    double finalResult;
    int averageIterationCount;

    public MultiStart(FreeParameters freeParameters){
        this.freeParameters = freeParameters;
        this.finalResult = Double.MAX_VALUE;
        this.averageIterationCount = 0;
    }
}
```


Класс содержит три поля, одно из которых является объектом, содержащим свободные параметры, и этот объект передается в конструктор.

- ***finalResult*** — поле, в которое записывается лучшее решение, найденное за все прогоны алгоритмов. В конструкторе присваивается максимальное вещественное значение по аналогичным предыдущему классу причинам
- ***averageIterationCount*** — поле, в которое записывается среднее значение итераций по всем запускам алгоритма. Этот параметр хорошо показывает интенсивность поиска. В конструкторе ему присваивается нулевое значение

```
public void multiStartRun() throws Exception{
    ExecutorService service = Executors.newWorkStealingPool();
    Future[] futureTasks = new Future[this.freeParameters.multiStartNumber];
    for(int start = 0; start < this.freeParameters.multiStartNumber; start++){
        final int startNumber = start;
        futureTasks[startNumber] = service.submit()->{
            StartAlgorithm running = new StartAlgorithm(freeParameters);
            running.run();
            if(running.overallBestFitnessFunctionValue < this.finalResult){
                this.finalResult = running.overallBestFitnessFunctionValue;
            }
            this.averageIterationCount += running.iterationCounter;
        };
    }
    for(Future task:futureTasks){
        task.get();
    }
    this.averageIterationCount /= this.freeParameters.multiStartNumber;
    System.out.print("Average iteration count: " + this.averageIterationCount + "\n");
}
```

Метод, позволяющий параллельно запустить несколько прогонов алгоритма. Объект со свободными параметрами содержит, кроме всего прочего, количество прогонов, введенное пользователем, и по этому значению в цикле с помощью инструментов для многопоточности в Java добавляются задания, которые состоят в инициализации класса ***StartAlgorithm*** и вызове метода ***run***. Далее, если полученное за алгоритм решение лучше записанного в поле ***finalResult***, то происходит перезапись. В конце к счетчику среднего количества итераций добавляется количество итераций данного прогона алгоритма. После завершения выполнения всех заданий счетчик делится на количество прогонов, и получается

средний показатель количества итераций. Этот показатель затем выводится на экран.

Класс *Interface*

Данный класс содержит метод ***main()***, с которого и начинается выполнение программы.

```
public class Interface {
    public static void main(String[] args) throws Exception{
        FreeParameters freeParameters = new FreeParameters();
        double startTime = System.nanoTime();
        MultiStart multiStart = new MultiStart(freeParameters);
        multiStart.multiStartRun();
        double executionTime = System.nanoTime() - startTime;
        DecimalFormat df = new DecimalFormat("#.#####");
        df.setRoundingMode(RoundingMode.CEILING);
        System.out.print("Minimum of the inserted function is " +
            df.format(multiStart.finalResult) + "\n");
        System.out.print("The exact value of function minimum is " +
            multiStart.finalResult + "\n");
        System.out.print("Execution time: " + executionTime / 1000000 + " ms " + "\n");
    }
}
```

Сначала происходит инициализация объекта, содержащего свободные параметры, и конструктор класса ***FreeParameters*** получит введенные пользователем значения. Затем вызывается метод ***MultiStart.multiStartRun()***, который выполняет определенное количество прогонов алгоритма и получает ответ, а также выводит среднее количество итераций. После этого в консоль также выводятся сам ответ, округленный до 10 знака после запятой, точное значение ответа для определения погрешности и время работы вызванного выше метода.

На этом работа программы завершается.

Тестирование работы алгоритмов

Значения свободных параметров

Для всех тестов использованы следующие значения свободных параметров:

- Число агентов популяции — 200
- Инерционная компонента — 0.7298 (рекомендованное значение)
- Когнитивная компонента — 1.49618 (рекомендованное значение)
- Социальная компонента — 1.49618 (рекомендованное значение)
- Максимальное количество итераций алгоритма — $2 \cdot 10^4$
- Минимальный предел достижения стагнации процесса — 10^2
- Количество запусков алгоритма — 10^2

Все тесты будут выполнены для размерностей пространства поиска $|X| = 2, 4, 8$

Критерии оценивания алгоритма

Будут использованы следующие критерии оценивания эффективности алгоритма:

- Лучшее по мультистарту значение целевой функции f^*

$$f^* = \min_{j \in 1:M} f_j^*,$$

где M — количество запусков алгоритма, f_j^* — лучшее значение целевой функции, полученное в алгоритме j .

- Среднее по мультистарту количество итераций T_{avg}

$$T_{avg} = \frac{1}{M} \sum_{j=1}^M T_j,$$

где T_j — количество итераций, проделанное алгоритмом j

- Средняя по мультистарту абсолютная погрешность решения Δf

$$\Delta f = \frac{1}{M} \sum_{j=1}^M |f_j^* - f^*|$$

- Вероятность локализации глобального минимума с точностью до $\Delta = 10^{-5} \ P$
- Время работы алгоритма

Тестирование алгоритма на различных функциях

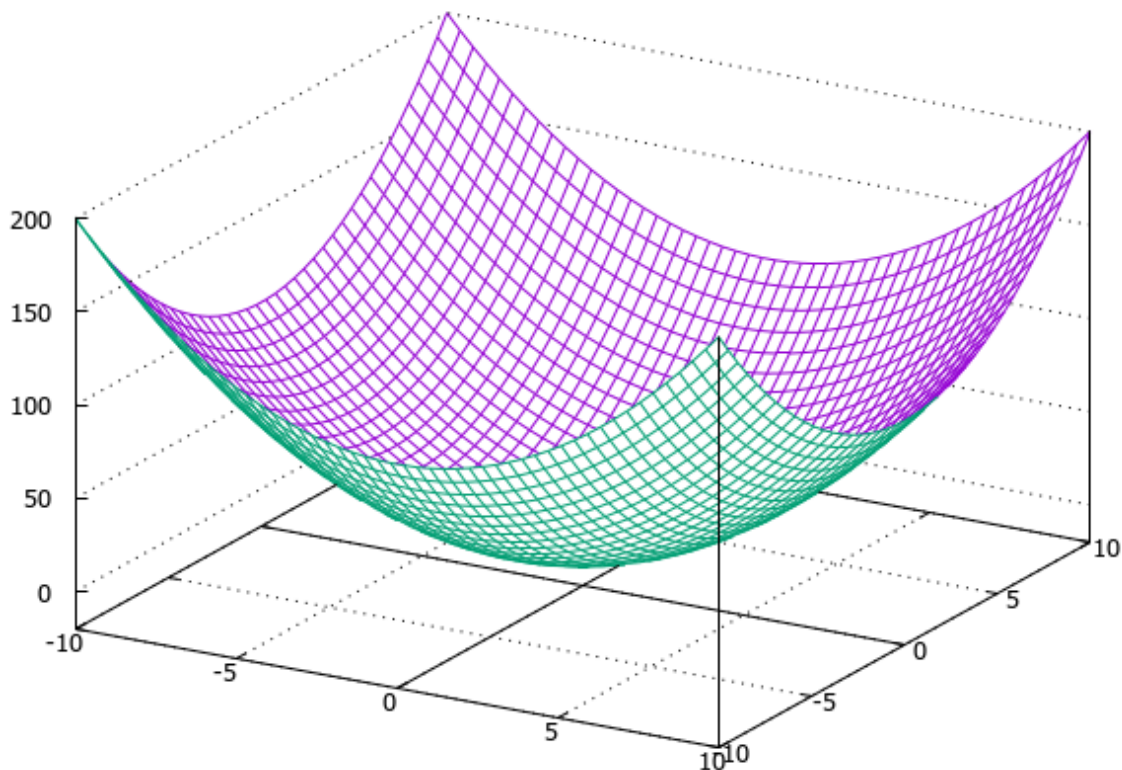
Функция сферы

$$f(X) = \sum_{i=1}^{|X|} x_i^2$$

Область поиска для данной функции:

$$D = \{X \in \mathbb{R}^{|X|} : |x_i| \leq 100 \quad \forall i \in 1 : |X|\}$$

Функция является унимодальной, имеет глобальный минимум в точке $X^* = (0, \dots, 0)$, минимальное значение $f^* = f(X^*) = 0$



функция сферы для $|X| = 2$

Реализация метода вычисления значений целевой функции

```
public double getFitnessFunctionValue(Vector X) {  
    double result = 0;  
    for(int i = 0; i < X.dimension; i++)  
        result += X.coordinates[i] * X.coordinates[i];  
    return result;  
}
```

Топология “кольцо”

Результаты вычислений алгоритма с использованием топологии кольца:

$ X $	f^*	Δf	T_{avg}	P	execution time
2	0	0	279	1	157 ms
4	0	0	486	1	1 sec 173 ms
8	0	0	1603	1	3 sec 433 ms

Топология “клика”

Результаты вычислений алгоритма с использованием топологии клики:

$ X $	f^*	Δf	T_{avg}	P	execution time
2	0	0	262	1	2 sec 777 ms
4	0	0	351	1	4 sec 283 ms
8	0	7.9e-12	440	0.9	7 sec 167 ms

Топология “двумерный тор”

Результаты вычислений алгоритма с использованием топологии двумерного тора:

$ X $	f^*	Δf	T_{avg}	P	execution time
2	0	0	349	1	783 ms
4	0	0	472	1	932 ms
8	0	0	680	1	1 sec 741 ms

Топология “кластер”

Результаты вычислений алгоритма с использованием кластерной топологии:

$ X $	f^*	Δf	T_{avg}	P	execution time
2	0	0	652	1	1 sec 208 ms
4	0	0	679	1	1 sec 700 ms
8	0	6e-10	1221	1	3 sec 292 ms

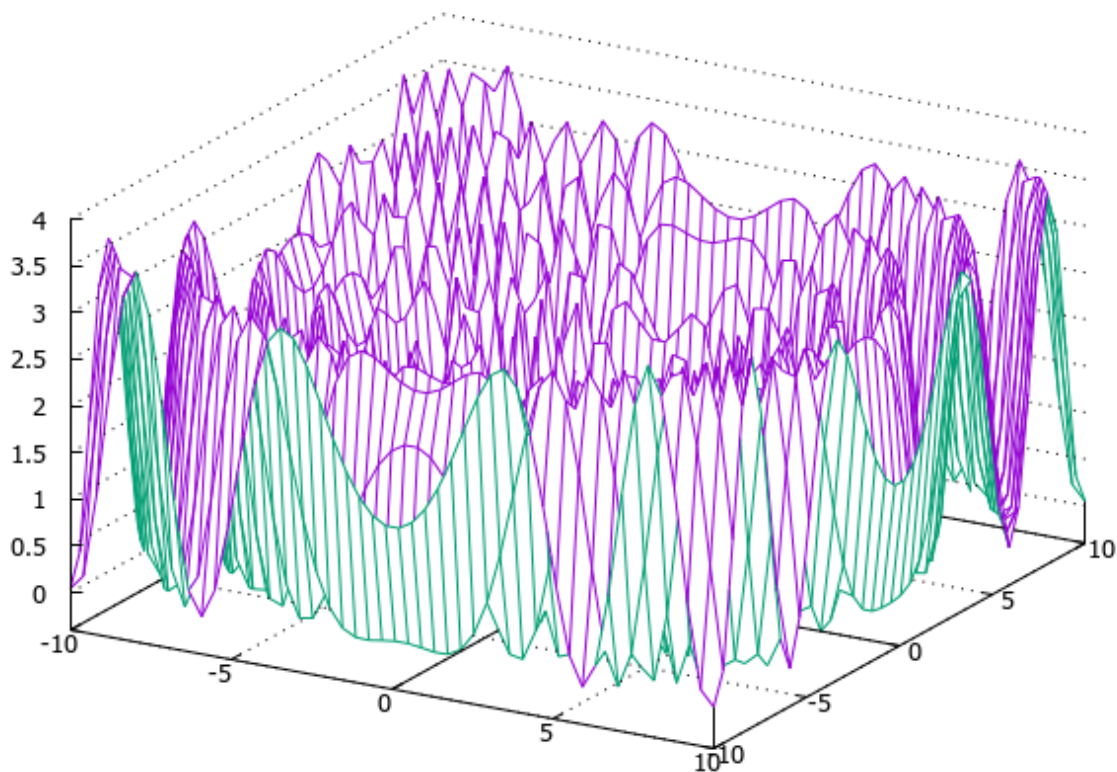
Функция Дэвиса

$$f(X) = (x_1^2 + x_2^2)^{0,25} \cdot (\sin^2(50 \cdot (x_1^2 + x_2^2)^{0,1}))$$

Область поиска для данной функции:

$$D = \{X \in \mathbb{R}^2 : |x_i| \leq 100 \quad \forall i \in 1 : |X| = 2\}$$

Функция имеет минимум в точке $X^* = (0, 0)$, минимальное значение $f^* = f(X^*) = 0$



функция Дэвиса

Реализация метода вычисления значений целевой функции

```
public double getFitnessFunctionValue(Vector X) {  
    double result = 0;  
    for(int i = 0; i < X.dimension; i++)  
        result += X.coordinates[i] * X.coordinates[i];  
    double tmp = result;  
    result = Math.pow(result, 0.25);  
}
```



```

    tmp = Math.pow(tmp, 0.1);
    tmp *= 50;
    tmp = Math.sin(tmp);
    tmp *= tmp;
    result *= tmp;
    return result;
}

```

Топология “кольцо”

Результаты вычислений алгоритма с использованием топологии кольца:

$ X $	f^*	Δf	T_{avg}	P	execution time
2	1	0	204	1	625 ms

Топология “клика”

Результаты вычислений алгоритма с использованием топологии клики:

$ X $	f^*	Δf	T_{avg}	P	execution time
2	1	0	243	1	12 sec 427 ms

Топология “двумерный тор”

Результаты вычислений алгоритма с использованием топологии двумерного тора:

$ X $	f^*	Δf	T_{avg}	P	execution time
2	1	0	198	1	708 ms

Топология “кластер”

Результаты вычислений алгоритма с использованием кластерной топологии:

$ X $	f^*	Δf	T_{avg}	P	execution time
2	1	0	234	1	1 sec 694 ms

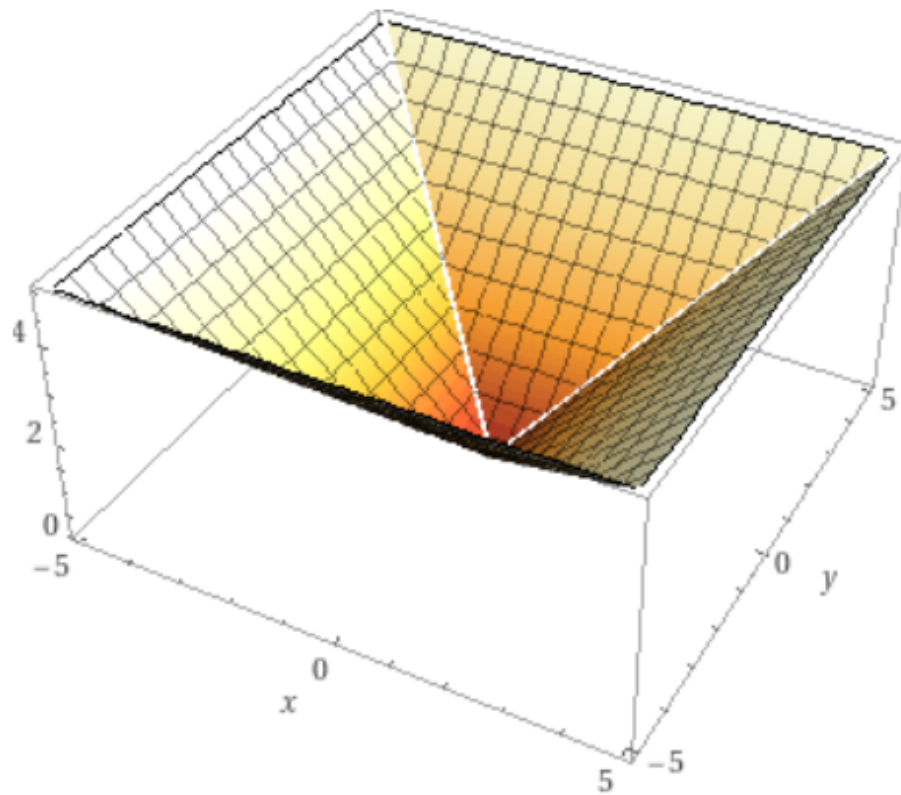
Функция Швевеля

$$f(X) = \max_i \{abs(x_i) : i \in 1 : |X|\}$$

Область поиска для данной функции:

$$D = \{X \in \mathbb{R}^{|X|} : |x_i| \leq 100 \quad \forall i \in 1 : |X|\}$$

Функция имеет минимум в точке $X^* = (0, \dots, 0)$, минимальное значение $f^* = f(X^*) = 0$



функция Швевеля для $|X| = 2$

Реализация метода вычисления значений целевой функции

```
public double getFitnessFunctionValue(Vector X) {  
    double result = Double.MIN_VALUE;  
    for(int i = 0; i < X.dimension; i++)  
        if(Math.abs(X.coordinates[i]) > result)  
            result = Math.abs(X.coordinates[i]);  
    return result;  
}
```

Топология “кольцо”

Результаты вычислений алгоритма с использованием топологии кольца:

$ X $	f^*	Δf	T_{avg}	P	execution time
2	0	0	3120	1	3 sec 462 ms
4	0	0	6790	1	10 sec 163 ms
8	0	0	19209	1	36 sec 927 ms

Топология “клика”

Результаты вычислений алгоритма с использованием топологии клики:

$ X $	f^*	Δf	T_{avg}	P	execution time
2	0	0	906	1	9 sec 885 ms
4	0	2.1e-9	384	1	4 sec 898 ms
8	0	4.7e-6	393	0.9	7 sec 178 ms

Топология “двумерный тор”

Результаты вычислений алгоритма с использованием топологии двумерного тора:

$ X $	f^*	Δf	T_{avg}	P	execution time
2	0	0	3313	1	4 sec 11 ms
4	0	0	4889	1	8 sec 430 ms
8	0	0	11835	1	27 sec 275 ms

Топология “кластер”

Результаты вычислений алгоритма с использованием кластерной топологии:

$ X $	f^*	Δf	T_{avg}	P	execution time
2	0	0	2838	1	4 sec 158 ms
4	0	0	3122	1	6 sec 7 ms
8	0	8e-7	16954	1	39 sec 696 ms

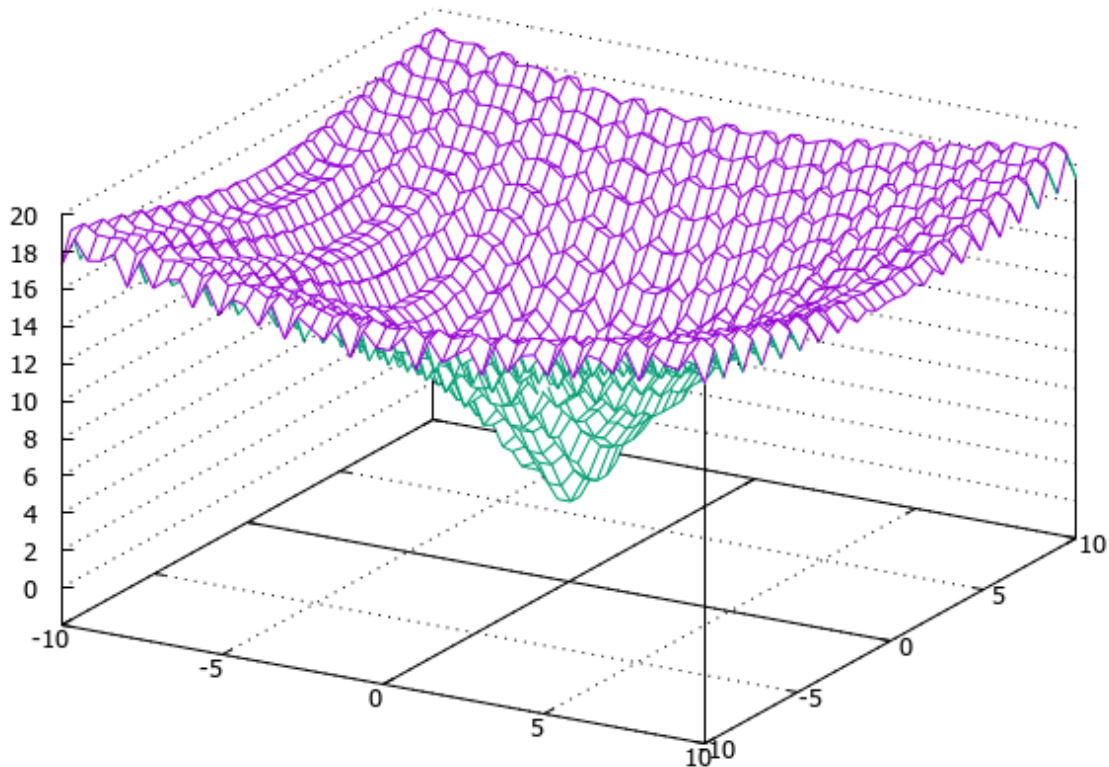
Функция Экли

$$f(X) = -20 \exp \left(-0,2 \sqrt{\frac{1}{|X|} \sum_{i=1}^{|X|} x_i^2} \right) - \exp \left(\frac{1}{|X|} \sum_{i=1}^{|X|} \cos(2\pi x_i) \right) + 20 + e$$

Область поиска для этой функции:

$$D = \{X \in \mathbb{R}^{|X|} : |x_i| \leq 32 \quad \forall i \in 1 : |X|\}$$

Функция имеет минимум в точке $X^* = (0, \dots, 0)$, минимальное значение $f^* = f(X^*) = 0$



функция Экли для $|X| = 2$

Реализация метода вычисления значений целевой функции

```
public double getFitnessFunctionValue(Vector X) {  
    double result = 0;  
    double tmp1 = 0;  
    double tmp2 = 0;
```

```

    for(int i = 0; i < X.dimension; i++){
        tmp1 += X.coordinates[i] * X.coordinates[i];
        tmp2 += Math.cos(2*Math.PI*X.coordinates[i]);
    }
    tmp1 /= X.dimension;
    tmp2 /= X.dimension;
    tmp1 = -0.2 * Math.sqrt(tmp1);
    tmp2 = Math.exp(tmp2);
    tmp1 = -20 * Math.exp(tmp1);
    result = tmp1 - tmp2 + 20 + Math.exp(1);
    return result;
}

```

Топология “кольцо”

Результаты вычислений алгоритма с использованием топологии кольца:

$ X $	f^*	Δf	T_{avg}	P	execution time
2	0	0	372	1	939 ms
4	0	0	530	1	1 sec 463 ms
8	0	1.1e-13	2154	1	7 sec 573ms

Топология “клика”

Результаты вычислений алгоритма с использованием топологии клики:

$ X $	f^*	Δf	T_{avg}	P	execution time
2	0	4.4e-16	332	1	15 sec 121 ms
4	0	1.4e-14	500	1	32 sec 233 ms
8	0	8.3e-6	635	0.7	64 sec 66 ms

Топология “двумерный тор”

Результаты вычислений алгоритма с использованием топологии двумерного тора:

$ X $	f^*	Δf	T_{avg}	P	execution time
2	0	4.4e-16	384	1	1 sec 26 ms
4	0	4.4e-16	497	1	1 sec 637 ms
8	0	4e-15	686	1	3 sec 263 ms

Топология “кластер”

Результаты вычислений алгоритма с использованием кластерной топологии:

$ X $	f^*	Δf	T_{avg}	P	execution time
2	0	4.4e-16	489	1	2 sec 859 ms
4	0	4.4e-16	652	1	4 sec 983 ms
8	0	4e-16	1222	1	15 sec 933 ms

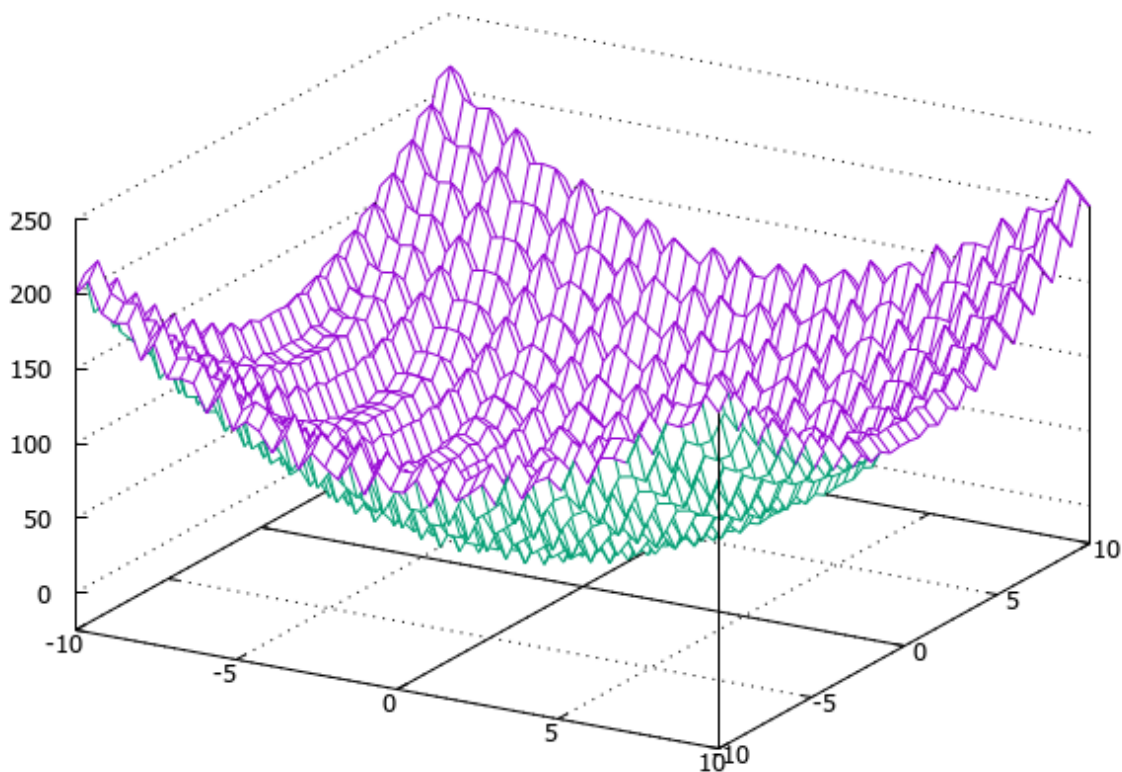
Функция Растригина

$$f(X) = \sum_{i=1}^{|X|} (x_i^2 - 10 \cos(2\pi x_i) + 10)$$

Область поиска для этой функции:

$$D = \{X \in \mathbb{R}^{|X|} : |x_i| \leq 5 \quad \forall i \in 1 : |X|\}$$

Функция имеет минимум в точке $X^* = (0, \dots, 0)$, минимальное значение $f^* = f(X^*) = 0$



функция Растригина для $|X| = 2$

Реализация метода вычисления значений целевой функции

```
public double getFitnessFunctionValue(Vector X) {  
    double result = 0;  
    for(int i = 0; i < X.dimension; i++){  
        result += X.coordinates[i] * X.coordinates[i];  
    }  
}
```



```

        result -= 10*Math.cos(2*Math.PI*X.coordinates[i]);
        result += 10;
    }
    return result;
}

```

Топология “кольцо”

Результаты вычислений алгоритма с использованием топологии кольца:

$ X $	f^*	Δf	T_{avg}	P	execution time
2	0	0	339	1	656 ms
4	0	9.9e-10	656	0.9	1 sec 396 ms
8	0	1.1e-6	2878	0.4	9 sec 27 ms

Топология “клика”

Результаты вычислений алгоритма с использованием топологии клики:

$ X $	f^*	Δf	T_{avg}	P	execution time
2	0	0	319	1	8 sec 344 ms
4	0	0	383	1	16 sec 434 ms
8	1.3	2.7	616	0	50 sec 411 ms

Топология “двумерный тор”

Результаты вычислений алгоритма с использованием топологии двумерного тора:

$ X $	f^*	Δf	T_{avg}	P	execution time
2	0	0	352	1	782 ms
4	0	0	479	0.95	1 sec 425 ms
8	0	4e-11	1000	0.35	4 sec 948 ms

Топология “кластер”

Результаты вычислений алгоритма с использованием кластерной топологии:

$ X $	f^*	Δf	T_{avg}	P	execution time
2	0	0	561	1	2 sec 16 ms
4	0	0	710	1	3 sec 899 ms
8	0	4e-11	1180	0.4	11 sec 703 ms

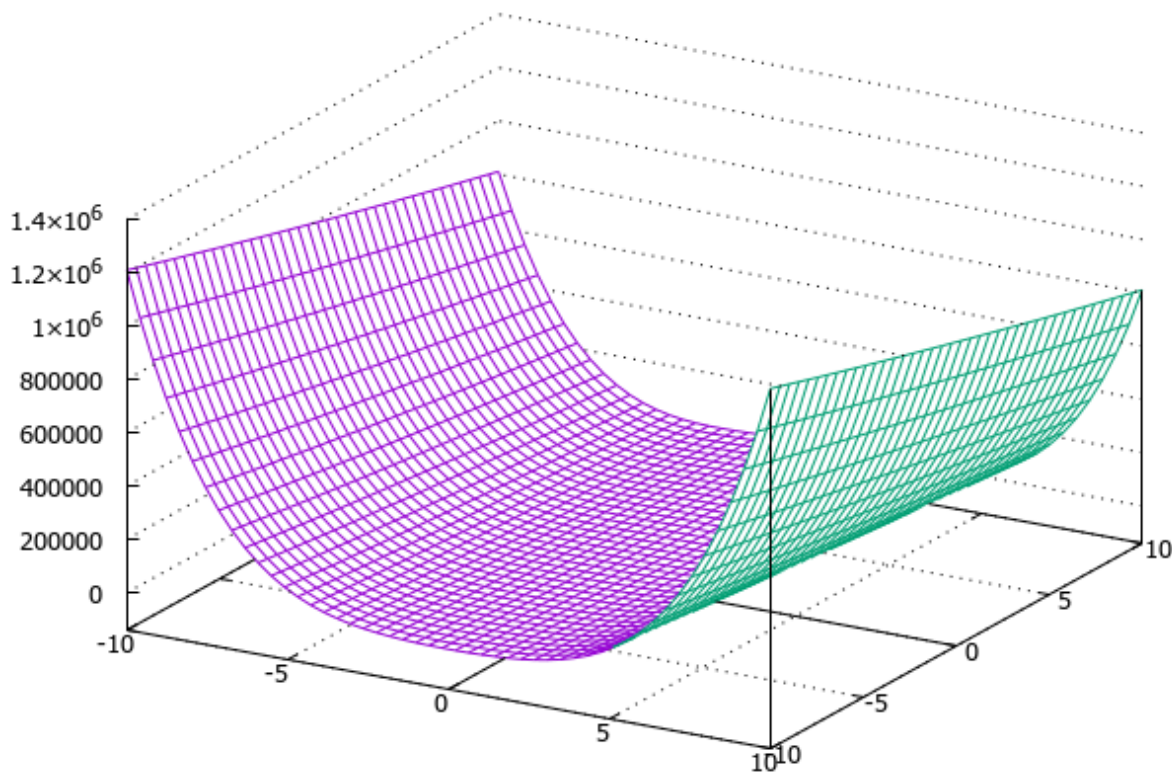
Функция Розенброка

$$f(X) = \sum_{i=1}^{|X|-1} (100(x_i^2 - x_{i+1})^2 + (x_i - 1)^2)$$

Область поиска для такой функции:

$$D = \{X \in \mathbb{R}^{|X|} : |x_i| \leq 100 \quad \forall i \in 1 : |X|\}$$

Функция имеет минимум в точке $X^* = (1, \dots, 1)$, максимальное значение $f^* = f(X^*) = 0$



функция Розенброка для $|X| = 2$

Реализация метода вычисления значений целевой функции

```
public double getFitnessFunctionValue(Vector X) {  
    double result = 0;  
    double tmp = 0;  
    for(int i = 0; i < X.dimension - 1; i++) {
```

```

        tmp = X.coordinates[i] * X.coordinates[i];
        tmp -= X.coordinates[i+1];
        tmp *= tmp;
        tmp *= 100;
        tmp += Math.pow((X.coordinates[i] - 1), 2);
        result += tmp;
    }
    return result;
}

```

Топология “кольцо”

Результаты вычислений алгоритма с использованием топологии кольца:

$ X $	f^*	Δf	T_{avg}	P	execution time
2	0	0	463	1	795 ms
4	0	0	763	1	1 sec 230 ms
8	0	0	3989	1	7 sec 859 ms

Топология “клика”

Результаты вычислений алгоритма с использованием топологии клики:

$ X $	f^*	Δf	T_{avg}	P	execution time
2	0	0	337	1	3 sec 99 ms
4	0	1.07e-9	357	0.8	4 sec 510 ms
8	1.6	3.1	412	0	21 sec 529 ms

Топология “двумерный тор”

Результаты вычислений алгоритма с использованием топологии двумерного тора:

$ X $	f^*	Δf	T_{avg}	P	execution time
2	0	0	523	1	789 ms
4	0	0	589	1	1 sec 69 ms
8	0	0	957	1	2 sec 231 ms

Топология “кластер”

Результаты вычислений алгоритма с использованием кластерной топологии:

$ X $	f^*	Δf	T_{avg}	P	execution time
2	0	0	746	1	1 sec 221 ms
4	0	0	609	1	1 sec 352 ms
8	0	0	1180	1	3 sec 220 ms

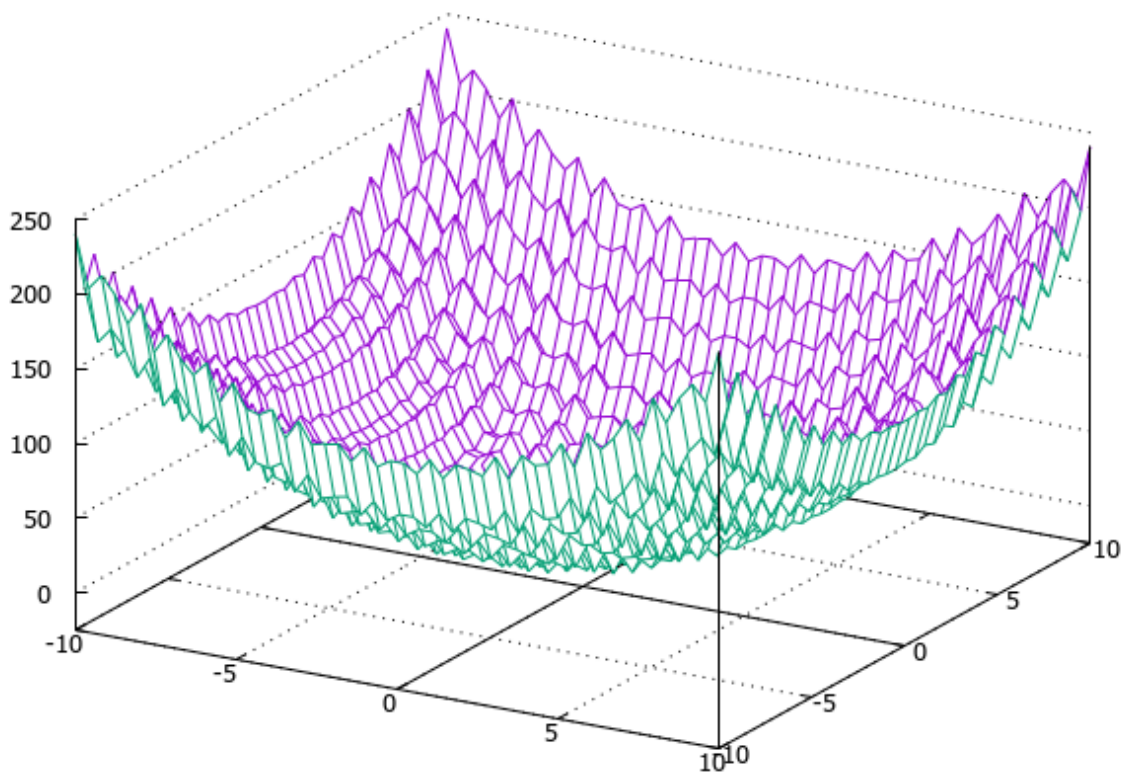
Многоэкстремальная функция

$$f(X) = \sum_{i=1}^{|X|} (x_i^2 + (|x_i| + 5) \cos(2\pi|x_i|) + 5,25)$$

Область поиска для такой функции:

$$D = \{X \in \mathbb{R}^{|X|} : |x_i| \leq 5 \quad \forall i \in 1 : |X|\}$$

Функция имеет $2^{|X|}$ глобальных минимумов в точках вида $X^* = (\pm 0,5, \pm 0,5, \dots, \pm 0,5)$, минимальное значение $f^* = f(X^*) = 0$



многоэкстремальная функция для $|X| = 2$

Реализация метода вычисления значений целевой функции

```
public double getFitnessFunctionValue(Vector X) {  
    double result = 0;  
    for(int i = 0; i < X.dimension; i++){  
        result += X.coordinates[i] * X.coordinates[i];  
    }  
}
```

```

        result += (Math.abs(X.coordinates[i]) + 5) *
        Math.cos(2 * Math.PI * X.coordinates[i]);
        result += 5.25;
    }
    return result;
}

```

Топология “кольцо”

Результаты вычислений алгоритма с использованием топологии кольца:

$ X $	f^*	Δf	T_{avg}	P	execution time
2	0	0	337	1	780 ms
4	0	0	580	1	1 sec 331 ms
8	0	0	2133	1	6 sec 640 ms

Топология “клика”

Результаты вычислений алгоритма с использованием топологии клики:

$ X $	f^*	Δf	T_{avg}	P	execution time
2	0	0	344	1	9 sec 30 ms
4	0	0	435	1	18 sec 694 ms
8	0	0.8	916	0.5	62 sec 529 ms

Топология “двумерный тор”

Результаты вычислений алгоритма с использованием топологии двумерного тора:

$ X $	f^*	Δf	T_{avg}	P	execution time
2	0	0	342	1	900 ms
4	0	0	493	1	1 sec 410 ms
8	0	0	737	1	3 sec 112 ms

Топология “кластер”

Результаты вычислений алгоритма с использованием кластерной топологии:

$ X $	f^*	Δf	T_{avg}	P	execution time
2	0	0	451	1	1 sec 651 ms
4	0	0	711	1	4 sec 186 ms
8	0	0	1156	1	11 sec 398 ms

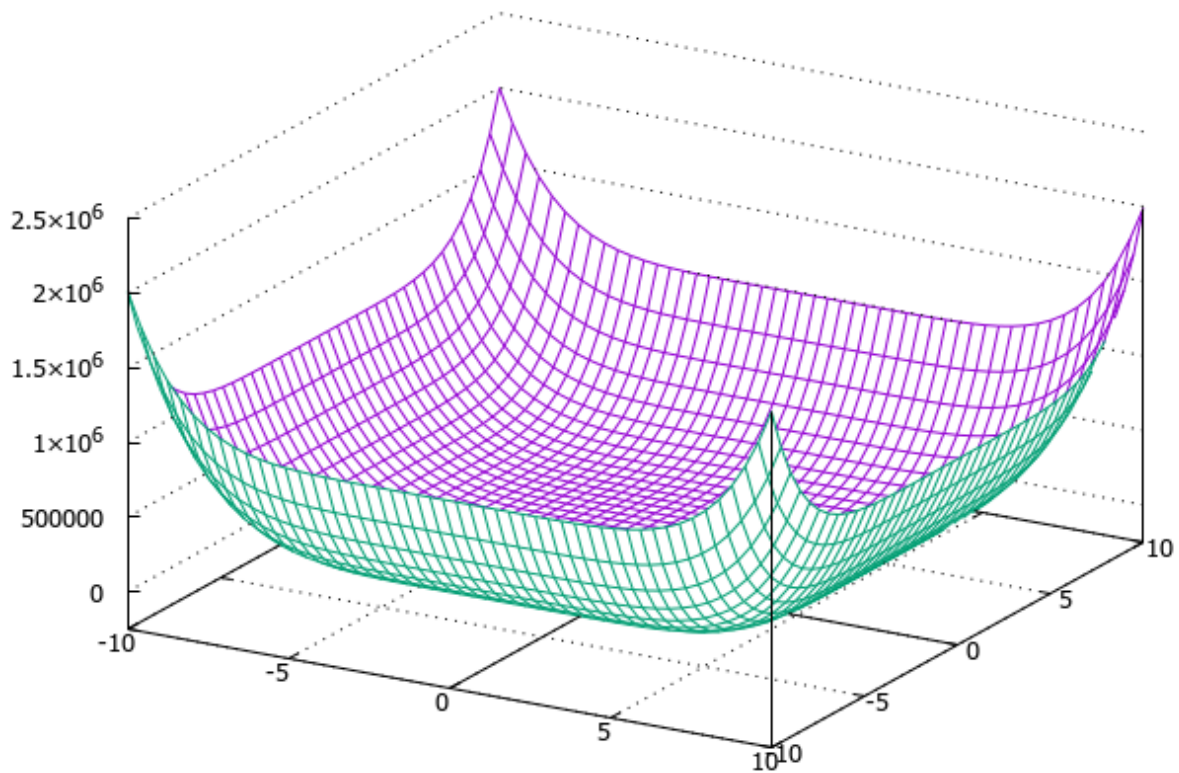
Полином, имеющий несколько локальных минимумов

$$f(X) = \sum_{i=1}^{|X|} (x_i^6 - 6x_i^3 - 6x_i^2 + 12x_i + 11)$$

Область поиска для такой функции:

$$D = \{X \in \mathbb{R}^{|X|} : |x_i| \leq 100 \quad \forall i \in 1 : |X|\}$$

Функция имеет минимум в точке $X^* = (-1, \dots, -1)$, минимальное значение $f^* = f(X^*) = 0$



полином для $|X| = 2$

Реализация метода вычисления значений целевой функции

```
public double getFitnessFunctionValue(Vector X) {  
    double result = 0;  
    for(int i = 0; i < X.dimension; i++){
```

```

        result += Math.pow(X.coordinates[i], 6);
        result -= 6 * Math.pow(X.coordinates[i], 3);
        result -= 6 * Math.pow(X.coordinates[i], 2);
        result += 12 * X.coordinates[i];
        result += 11;
    }
    return result;
}

```

Топология “кольцо”

Результаты вычислений алгоритма с использованием топологии кольца:

$ X $	f^*	Δf	T_{avg}	P	execution time
2	0	0	400	1	1 sec 91 ms
4	0	0	542	1	2 sec 352 ms
8	0	7.66e-8	2530	1	19 sec 928 ms

Топология “клика”

Результаты вычислений алгоритма с использованием топологии клики:

$ X $	f^*	Δf	T_{avg}	P	execution time
2	0	1.6e-10	438	1	29 sec 229 ms
4	0	4.6e-10	518	1	69 sec 920 ms
8	0	0.27	741	0.4	191 sec 678 ms

Топология “двумерный тор”

Результаты вычислений алгоритма с использованием топологии двумерного тора:

$ X $	f^*	Δf	T_{avg}	P	execution time
2	0	0	388	1	1 sec 383 ms
4	0	4.6e-10	487	1	2 sec 909 ms
8	0	1.3e-9	792	1	8 sec 331 ms

Топология “кластер”

Результаты вычислений алгоритма с использованием кластерной топологии:

$ X $	f^*	Δf	T_{avg}	P	execution time
2	0	5.9e-10	558	1	5 sec 48 ms
4	0	3e-9	674	1	10 sec 450 ms
8	0	4e-9	1046	1	32 sec 703 ms

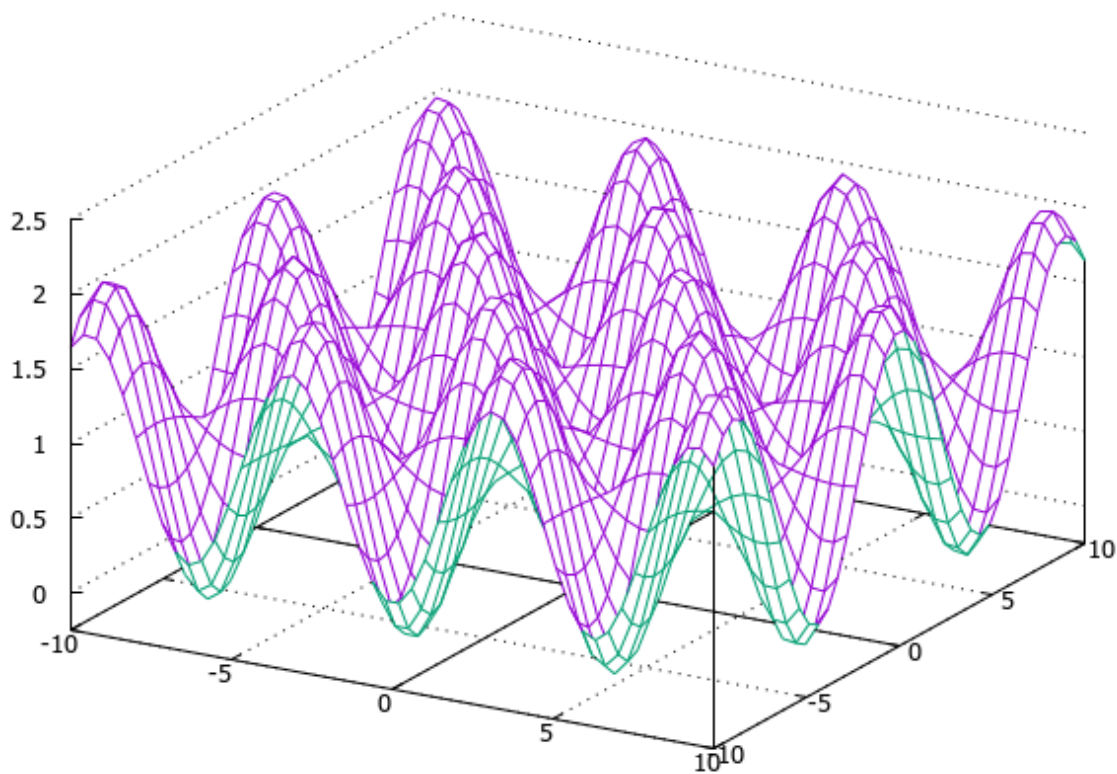
Функция Гриванка

$$f(X) = \sum_{i=1}^{|X|} \frac{x_i^2}{4000} - \prod_{i=1}^{|X|} \cos\left(\frac{x_i}{\sqrt{i}}\right) + 1$$

Область поиска для такой функции:

$$D = \{X \in \mathbb{R}^{|X|} : |x_i| \leq 16 \quad \forall i \in 1 : |X|\}$$

Функция имеет минимум в точке $X^* = (0, \dots, 0)$, минимальное значение $f^* = f(X^*) = 0$



функция Гриванка для $|X| = 2$

Реализация метода вычисления значений целевой функции

```
public double getFitnessFunctionValue(Vector X) {  
    double result = 0;  
    double prod = 1;  
    for(int i = 0; i < X.dimension; i++){
```

```

        result += X.coordinates[i] * X.coordinates[i];
        prod *= Math.cos(X.coordinates[i] / Math.sqrt(i + 1));
    }
    result /= 4000;
    result -= prod;
    result++;
    return result;
}

```

Топология “кольцо”

Результаты вычислений алгоритма с использованием топологии кольца:

$ X $	f^*	Δf	T_{avg}	P	execution time
2	0	0	353	1	748 ms
4	0	4.5e-8	451	0.98	1 sec 360 ms
8	0	7.66e-5	1548	0.8	5 sec 533 ms

Топология “клика”

Результаты вычислений алгоритма с использованием топологии клики:

$ X $	f^*	Δf	T_{avg}	P	execution time
2	0	2.7e-6	332	0.95	10 sec 292 ms
4	0.019	3.6e-1	500	0	25 sec 10 ms
8	0.025	0.81	1050	0	110 sec 396 ms

Топология “двумерный тор”

Результаты вычислений алгоритма с использованием топологии двумерного тора:

$ X $	f^*	Δf	T_{avg}	P	execution time
2	0	0	328	1	760 ms
4	0	0	530	1	1 sec 626 ms
8	0	0	600	1	3 sec 14 ms

Топология “кластер”

Результаты вычислений алгоритма с использованием кластерной топологии:

$ X $	f^*	Δf	T_{avg}	P	execution time
2	0	0	409	1	1 sec 766 ms
4	0	0	627	1	4 sec 160 ms
8	0	4e-6	846	0.95	10 sec 816 ms

Вывод

Проанализирована эффективность алгоритмов роя частиц с разными топологиями — “кольцо”, “клика”, “двумерный тор” и “кластер”. Результаты показывают, что глобальные решения оптимизации каждой тестовой функции были найдены всеми алгоритмами, кроме алгоритма, использующего топологию “клика”.

Алгоритм, использующий топологию “клика”, дал самые высокие показатели интенсивности сходимости, что объясняет отсутствие выявленного решения у некоторых функций при некоторых размерностях. Такой алгоритм плохо подходит под решение многоэкстримальный задач оптимизации, а его в силу большого количества соседей вычислительная сложность у него выше, чем у остальных алгоритмов. Очевидно, что в общем случае такой алгоритм использовать нецелесообразно.

Алгоритм, использующий топологию “кластер”, показал более высокие результаты, обнаружив глобальные минимумы у каждой функции. Интенсивность сходимости этого алгоритма не так велика, но вероятность локализации глобального экстремума в большинстве тестовых задач оставляет желать лучшего. Такой алгоритм лучше подходит для решения многоэкстримальных задач, однако его использование может потребовать многократный запуск для выявления настоящего решения и в случае, если это решение неизвестно, алгоритм может привести к неправильному ответу.

Алгоритм, использующий топологию “кольцо”, продемонстрировал хорошую вероятность вычисления глобального минимума функции, и в силу небольшого количества соседей его вычислительная сложность невелика. Однако при увеличении размерности пространства скорость сходимости будет увеличиваться нелинейно, то есть быстрее увеличения размерности. Это приведет к резкому увеличению количества вычислений и замедлит работу алгоритма. Из плюсов данного метода можно ответить большую ширину поиска.

Алгоритм, использующий топологию “двумерный тор”, показал наилучшие результаты среди всех алгоритмов. Имея самые высокие вероятности локализации и самые низкие погрешности целевых функций, алгоритм имеет достаточно низкую вычислительную сложность, которая с увеличением размерности пространства растет не так быстро, как сложность алгоритма с топологией “кольцо”. Ко всему прочему, этот алгоритм продемонстрировал хороший баланс между

интенсивностью и диверсификацией поиска, а также обогнал почти все алгоритмы по затраченному на вычисления времени.

Подводя итог всему вышесказанному, можно утверждать, что наиболее оптимальным решением при выборе алгоритма оптимизации среди представленных четырех будет алгоритм с топологией “двумерный тор”. Он хорошо работает как на простых функциях, так и на многоэкстремальных вроде представленной выше **многоэкстремальной функции**.

Заключение

В данной работе был рассмотрен канонический алгоритм роя частиц, используемый для оптимизации многокритериальных функций, а также четыре топологии соседства частиц. Все четыре алгоритма были реализованы на языке Java и затем протестированы на различных функциях. По итогам тестов получилось, что топология двумерного тора является наиболее оптимальной для выбора при отсутствии каких-либо наперед известных свойств задачи оптимизации, которые могут повлиять на выбор.

Литература

- А. П. Карпенко — “Современные алгоритмы поисковой оптимизации. Алгоритмы, вдохновленные природой” (Москва, издательство МГТУ имени Н. Э. Баумана, 2017);
- А. Б. Сергиенко — “Тестовые функции для глобальной оптимизации” (Красноярск, Сибирский Государственный Аэрокосмический Университет имени М. Ф. Решетнева, 2015);