МИНОБРНАУКИ РОССИИ
Федеральное государственное бюджетное образовательное учреждение высшего образования
"МИРЭА - Российский технологический университет"
РТУ МИРЭА

Институт Информационных Технологий

Кафедра Вычислительной Техники

ПРАКТИЧЕСКАЯ РАБОТА №3

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

Студент группы:ИКБО-04-22 __Кликушин В.И._
(Ф. И.О. студента)

Преподаватель __Железняк Л.М.__
(Ф.И.О. преподавателя)

Москва 2024

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Алгоритм роя частиц был разработан в середине 1990-х годов учеными Расселом Эберхартом и Джеймсом Кеннеди, которые вдохновились наблюдениями за коллективным поведением стай птиц и косяков рыб. В процессе поиска пищи такие группы стремятся к общей цели, используя взаимодействуя друг с другом, что позволяет им находить оптимальные маршруты и избегать опасностей. Этот феномен коллективного интеллекта показал, что сложное поведение может возникать из простых взаимодействий. Интерес алгоритма заключается в его способности моделировать такое поведение для решения задач оптимизации, где каждое «решение» представлено частицей, а весь процесс — роем, который перемещается в поисках оптимума.

Алгоритм роя частиц является мощным инструментом для решения широкого круга задач численной и комбинаторной оптимизации. Благодаря своей гибкости и способности эффективно находить глобальные минимумы, он находит применение во многих научно-технических областях.

В управлении энергетическими системами алгоритм помогает в оптимальном распределении ресурсов и управлении нагрузками, что повышает общую устойчивость и снижает затраты. В задачах календарного планирования рой частиц позволяет разрабатывать оптимальные графики, учитывающие множество ограничений и требований. В сфере мобильной связи алгоритм используется для оптимизации сетевых параметров. Помимо этого, алгоритм роя частиц активно применяется в обработке изображений и распознавании образов, помогая улучшать качество изображений и точность классификации.

Благодаря возможности эффективно исследовать пространство решений, алгоритм роя частиц успешно справляется с задачами, где традиционные методы оказываются менее эффективными. Широкая применимость делает его ценным инструментом для многих научных и инженерных приложений.

1 АЛГОРИТМ РОЯ ЧАСТИЦ

Алгоритм роя является методом численной оптимизации, поддерживающий общее количество возможных решений, которые называются частицами или агентами, и перемещая их в пространстве к наилучшему найденному в этом пространстве решению, всё время находящемуся в изменении из-за нахождения агентами более выгодных решений.

1.1 Описание алгоритма

Роевый алгоритм использует рой частиц, где каждая частица представляет потенциальное решение проблемы. Поведение частицы в гиперпространстве поиска решения все время подстраивается в соответствии со своим опытом и опытом своих соседей. Кроме этого, каждая частица помнит свою лучшую позицию с достигнутым локальным лучшим значением целевой функции и знает наилучшую позицию частиц - своих соседей, где достигнут глобальный на текущий момент оптимум. В процессе поиска частицы роя обмениваются информацией о достигнутых лучших результатах и изменяют свои позиции и скорости по определенным правилам на основе имеющейся на текущий момент информации о локальных и глобальных достижениях. При этом глобальный лучший результат известен всем частицам и немедленно корректируется в том случае, когда некоторая частица роя находит лучшую позицию с результатом, превосходящим текущий глобальный оптимум.

Обобщая выше сказанное, алгоритм состоит из следующих ключевых шагов:

1. Создание роя частиц.
2. Нахождение лучшего решения для каждой частицы.
3. Нахождение лучшего решения для всех частиц.
4. Коррекция скорости каждой частицы (Формула 1.3.5).
5. Перемещение каждой частицы (Формула 1.3.4).

6. Завершение работы, если критерий останова выполнен, иначе возврат к пункту номер два.

1.2 Постановка задачи

Цель работы: реализовать преобразование Коши методом роя частиц для нахождения приближённого глобального минимума целевой функции.

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.

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

Выбранная функция для оптимизации: функция Гольдшейна-Прайса (Формула 1.2.1).

$$f(x,y) = [1 + (x + y + 1)^2(19 - 14x + 3x^2 - 14y + 6xy + 3y^2)][30 + \\ +(2x - 3y)^2(18 - 32x + 12x^2 + 48y - 36xy + 27y^2)] \quad (1.2.1)$$

Глобальный минимум функции достигается в точке (0; -1) и равен 3. Функция рассматривается на области $-2 \le x, y \le 2$.

1.3 Математическая модель

Начальная популяция состоит из N частиц. Каждая частица описывается вектором координат и вектором скоростей (Формула 1.3.1).

$$\underset{x}{\rightarrow}{}^{k}_{j} = (x^k_{j1}, x^k_{j2}, \dots, x^k_{ji}, x^k_{jn}); \underset{v}{\rightarrow}{}^{k}_{j} = (v^k_{j1}, v^k_{j2}, \dots, v^k_{ji}, v^k_{jn}); j = 1 \dots N; i = 1 \dots n, \quad (1.3.1)$$

где x – координата частицы в гиперпространстве;

j – номер частицы;

k – номер итерации алгоритма;

i – номер координаты;

v – скорость координаты;

N – общее количество частиц;

n – количество измерений.

Лучшее решение для j-й частицы на итерациях $k = 0 \dots m$ обозначено в соответствии с Формулой 1.3.2.

$$\underset{p}{\rightarrow}{}_{j}^{k} = (p_{j1}^{k}, p_{j2}^{k}, \dots, p_{ji}^{k}, p_{jn}^{k}) \quad (1.3.2)$$

Лучшее решение для всех частиц на итерациях $k = 0 \dots m$ обозначено в соответствии с Формулой 1.3.3.

$$\underset{b}{\rightarrow}{}^{k} = (b_{1}^{k}, b_{2}^{k}, \dots, b_{i}^{k}, b_{n}^{k}) \quad (1.3.3)$$

Перемещение частицы осуществляется в соответствии с Формулой 1.3.4.

$$x_{ji}^{k+1} = x_{ji}^{k} + v_{ji}^{k+1} \quad (1.3.4)$$

Вектор скорости управляет процессом поиска решения и его компоненты определяются с учетом когнитивной и социальной составляющей. На каждой $k + 1$ итерации для j-й частицы i-я компонента скорости определяется в соответствии с Формулой 1.3.5.

$$v_{ji}^{k+1} = v_{ji}^{k} + a_{p} r_{p} (p_{ji}^{k} - x_{ji}^{k}) + a_{b} r_{b} (b_{i}^{k} - x_{ji}^{k});\ i = 1, \dots n;\ j = 1, \dots N, \quad (1.3.5)$$

где a_p – подбираемый положительный коэффициент ускорения;

r_p – случайное число из диапазона [0;1], которое вносит элемент случайности в процесс поиска;

a_b – подбираемый положительный коэффициент ускорения;

r_b – случайное число из диапазона [0;1], которое вносит элемент случайности в процесс поиска.

1.4 Глобальный роевой алгоритм

Существует два основных подхода в оптимизации роя частиц, под названиями lbest и gbest, отличающиеся топологией соседства, используемой для обмена опытом между частицами.

1.4.1 Особенность реализации

При коррекции скорости частицы используется информация о положении достигнутого глобального оптимума, которая определяется на основании информации, передаваемой всеми частицами роя, то есть для модели gbest лучшая частица определяется из всего роя. Это классическая и наиболее популярная версия алгоритма.

На Рисунке 1.4.1 представлена структура «звезда», где все частицы связаны друг с другом (образуют полный граф) и могут соответственно обмениваться информацией.

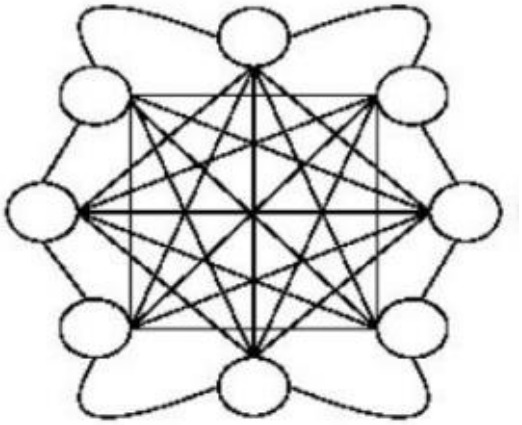

Рисунок 1.4.1 – Структура сети «звезда»

В этом случае каждая частица стремится сместиться в сторону глобальной лучшей позиции, которую нашел рой. Основной роевый алгоритм (глобальный) использует по умолчанию фактически структуру «звезда» на всем рое.

1.4.2 Ручной расчёт

Подбираемые коэффициенты ускорения приняты равными 2.

Случайным образом сгенерированы начальные координаты и компоненты скоростей для трёх частиц.

Первая частица. Координаты: (-0.0842, 0.4813); Скорость: (0.6218, -0.6931); Лучшее значение функции: 4619.98.

Вторая частица. Координаты: (-1.6099, 1.9618); Скорость: (-0.9485, -0.6894); Лучшее значение функции: 937558.12.

Третья частица. Координаты: (1.0100, -1.7479); Скорость: (-0.9026, -0.0767); Лучшее значение функции: 10001.47.

На первой итерации первая частица перемещается в соответствие с Формулами 1.4.2.1-1.4.2.2.

$$x_{11}^{1} = x_{11}^{0} + v_{11}^{0} = -0.0842 + 0.6219 = 0.5377; \quad (1.4.2.1)$$

$$x_{12}^{1} = x_{12}^{0} + v_{12}^{0} = 0.4813 - 0.6931 = -0.2118 \quad (1.4.2.2)$$

Значение функции в полученной точке равно 761.44. Получено лучшее решение, значит для первой частицы переписывается вектор координат и лучшее значение функции (в пределах частицы).

Обновляются координаты и значение глобального минимума для всего роя.

На первой итерации вторая частица перемещается в соответствие с Формулами 1.4.2.3-1.4.2.4.

$$x_{21}^1 = x_{21}^0 + v_{21}^0 = -1.6099 - 0.9485 = -2; \quad (1.4.2.3)$$

$$x_{22}^1 = x_{22}^0 + v_{22}^0 = 1.9618 - 0.6894 = 1.2723 \quad (1.4.2.4)$$

Так как для первой компоненты получена координата, выходящая за пределы диапазона поиска, её значением становится нижняя граница диапазона -2.

Значение фитнесс-функции в полученной точке: 65561.66. Получено лучшее решение, значит для второй частицы переписывается вектор координат и лучшее значение функции (в пределах частицы).

Лучшее глобальное решение остаётся без изменений.

На первой итерации третья частица перемещается в соответствие с Формулами 1.4.2.5-1.4.2.6.

$$x_{31}^1 = x_{31}^0 + v_{31}^0 = 1.0100 - 0.9026 = 0.1073; \quad (1.4.2.5)$$

$$x_{32}^1 = x_{32}^0 + v_{32}^0 = -1.7479 - 0.0767 = -1.8246 \quad (1.4.2.6)$$

Значение функции в полученной точке равно 22400.52. Получено худшее решение, значит для третьей частицы сохраняется вектор координат и значение функции в этой точке.

Лучшее глобальное решение остаётся без изменений.

На первой итерации компоненты вектора скорости для первой частицы обновляются в соответствие с Формулами 1.4.2.7-1.4.2.8. Сгенерированные случайные числа: [0.2445, 0.8954]; [0.8667, 0.0035].

$$v_{11}^1 = v_{11}^0 + a_p r_p (p_{11}^1 - x_{11}^1) + a_b r_b (b_1^1 - x_{11}^1) = 0.6219 + 2 * 0.2445 * \\ * (0.5377 - 0.5377) + 2 * \ 0.8954 * (0.5377 - 0.5377) = 0.6218; \quad (1.4.2.7)$$

$$v_{12}^1 = v_{12}^0 + a_p r_p(p_{12}^1 - x_{12}^1) + a_b r_b(b_2^1 - x_{12}^1) = -0.6931 + 2 * 0.8667 *$$
$$* (-0.2118 + 0.2118) + 2 * \ 0.0035 * (-0.2118 + 0.2118) = -0.6931 \quad (1.4.2.8)$$

На первой итерации компоненты вектора скорости для второй частицы обновляются в соответствие с Формулами 1.4.2.9-1.4.2.10. Сгенерированные случайные числа: [0.1084,0.5134]; [0.9882,0.2198].

$$v_{21}^1 = v_{21}^0 + a_p r_p(p_{21}^1 - x_{11}^1) + a_b r_b(b_1^1 - x_{21}^1) = -0.9485 + 2 * 0.1084 *$$
$$* (-2 + 2) + 2 * \ 0.5134 * (0.5377 + 2) = 1.6573; \quad (1.4.2.9)$$

$$v_{22}^1 = v_{22}^0 + a_p r_p(p_{22}^1 - x_{22}^1) + a_b r_b(b_2^1 - x_{22}^1) = -0.6894 + 2 * 0.9882 *$$
$$* (1.2723 - 1.2723) + 2 * \ 0.2198 * (-0.2118 - 1.2723) = -1.3420 \quad (1.4.2.10)$$

На первой итерации компоненты вектора скорости для третьей частицы обновляются в соответствие с Формулами 1.4.2.11-1.4.2.12. Сгенерированные случайные числа: [0.0267,0.3430]; [0.4052,0.5252].

$$v_{31}^1 = v_{31}^0 + a_p r_p(p_{31}^1 - x_{31}^1) + a_b r_b(b_1^1 - x_{31}^1) = -0.9025 + 2 * 0.0267 *$$
$$* (1.0099 - 0.1073) + 2 * \ 0.3430 * (0.5377 - 0.1073) = -0.5589; \quad (1.4.2.11)$$

$$v_{32}^1 = v_{32}^0 + a_p r_p(p_{32}^1 - x_{32}^1) + a_b r_b(b_2^1 - x_{32}^1) = -0.0767 + 2 * 0.4052 *$$
$$* (-1.7479 + 1.8246) + 2 * \ 0.5252 * (-0.2118 + 1.8246) = 1.6798 \quad (1.4.2.12)$$

Под конец первой итерации лучшая точка минимума: (0.5377; -0.2118). Значение функции в этой точке: 761.44.

На второй итерации первая частица перемещается в соответствие с Формулами 1.4.2.13-1.4.2.14.

$$x_{11}^2 = x_{11}^1 + v_{11}^1 = 0.5377 + 0.6218 = 1.1596; \quad (1.4.2.13)$$

$$x_{12}^2 = x_{12}^1 + v_{12}^1 = -0.2118 - 0.6930 = -0.9049 \quad (1.4.2.14)$$

Значение функции в полученной точке равно 9514.66. Получено худшее решение, значит для первой частицы вектор координат и лучшее значение функции сохраняются без изменений (в пределах частицы).

На второй итерации вторая частица перемещается в соответствие с Формулами 1.4.2.15-1.4.2.16.

$$x_{21}^2 = x_{21}^1 + v_{21}^1 = -2 + 1.6573 = -0.3426; \quad (1.4.2.15)$$

$$x_{22}^2 = x_{22}^1 + v_{22}^1 = 1.2723 - 1.3420 = -0.0697 \quad (1.4.2.16)$$

Значение функции в полученной точке равно 349.83. Получено лучшее решение, значит для второй частицы вектор координат и лучшее значение функции переписываются (в пределах частицы).

Обновляются глобальная лучшая точка и значение функции в этой точке для всего роя.

На второй итерации третья частица перемещается в соответствие с Формулами 1.4.2.17-1.4.2.18.

$$x_{31}^2 = x_{31}^1 + v_{31}^1 = 0.1073 - 0.5589 = -0.4515; \quad (1.4.2.17)$$

$$x_{32}^2 = x_{32}^1 + v_{32}^1 = -1.8246 + 1.6798 = -0.1448 \quad (1.4.2.18)$$

Значение функции в полученной точке равно 201.20. Получено лучшее решение, значит для третьей частицы вектор координат и лучшее значение функции переписываются (в пределах частицы).

Обновляются глобальная лучшая точка и значение функции в этой точке для всего роя.

На второй итерации компоненты вектора скорости для первой частицы обновляются в соответствие с Формулами 1.4.2.19-1.4.2.20. Сгенерированные случайные числа: [0.3548, 0.4196]; [0.6361, 0.7756].

$$v_{11}^2 = v_{11}^1 + a_p r_p(p_{11}^2 - x_{11}^2) + a_b r_b(b_1^2 - x_{11}^2) = 0.6218 + 2 * 0.3548 * \\ * (0.5377 - 1.1596) + 2 * \ 0.4196 * (-0.4515 - 1.1596) = -1.1718; \quad (1.4.2.19)$$

$$v_{12}^2 = v_{12}^1 + a_p r_p(p_{12}^2 - x_{12}^2) + a_b r_b(b_2^2 - x_{12}^2) = -0.6930 + 2 * 0.6361 * \\ * (-0.2118 + 0.9049) + 2 * \ 0.7756 * (-0.1448 + 0.9049) = 1.3678 \quad (1.4.2.20)$$

На второй итерации компоненты вектора скорости для второй частицы обновляются в соответствие с Формулами 1.4.2.21-1.4.2.22. Сгенерированные случайные числа: [0.1800, 0.2495]; [0.0777, 0.1417].

$$v_{21}^2 = v_{21}^1 + a_p r_p(p_{21}^2 - x_{21}^2) + a_b r_b(b_1^2 - x_{21}^2) = 1.6573 + 2 * 0.1800 * \\ * (-0.3426 + 0.3426) + 2 * \ 0.2495 * (-0.4515 + 0.3426) = 1.6029; \quad (1.4.2.21)$$

$$v_{22}^2 = v_{22}^1 + a_p r_p(p_{22}^2 - x_{22}^2) + a_b r_b(b_2^2 - x_{22}^2) = -1.3420 + 2 * 0.0777 * \\ * (-0.0697 + 0.0697) + 2 * \ 0.1417 * (-0.1448 + 0.0697) = -1.3633 \quad (1.4.2.22)$$

На второй итерации компоненты вектора скорости для третьей частицы обновляются в соответствие с Формулами 1.4.2.23-1.4.2.24. Сгенерированные случайные числа: [0.8495, 0.2512]; [0.3009, 0.9099].

$$v_{31}^2 = v_{31}^1 + a_p r_p(p_{31}^2 - x_{31}^2) + a_b r_b(b_1^2 - x_{31}^2) = -0.5589 + 2 * 0.8495 * \\ * (-0.4515 + 0.4515) + 2 * \ 0.2512 * (-0.4515 + 0.4515) = -0.5589; \quad (1.4.2.23)$$

$$v_{32}^2 = v_{32}^1 + a_p r_p(p_{32}^2 - x_{32}^2) + a_b r_b(b_2^2 - x_{32}^2) = 1.6798 + 2 * 0.3009 *$$
$$* (-0.1448 + 0.1448) + 2 * \ 0.9099 * (-0.1448 + 0.1448) = 1.6798 \quad (1.4.2.24)$$

По окончании второй итерации координаты точки глобального минимума: (-0.4515; -0.1448). Значение функции в этой точке: 201.20.

1.5 Локальный роевый алгоритм

1.5.1 Особенность реализации

Локальный роевый алгоритм использует для коррекции вектора скорости частицы только локальный оптимум, который определяется на множестве соседних (ближайших в некотором смысле) частиц. То есть считается, что данной частице может передавать полезную информацию только ее ближайшее окружение. При этом отношение соседства задается некоторой «социальной» сетевой структурой, которая образует перекрывающееся множества соседних частиц, которые могут влиять друг на друга. Соседние частицы обмениваются между собой информацией о достигнутых лучших результатах и поэтому стремятся двигаться в сторону локального в данной окрестности оптимума.

Выбранная социальная сеть: кольцо. Каждая частица обменивается информацией с двумя соседними частицами. В таком случае каждая частица стремится сместиться в сторону лучшего соседа. Социальная структура кольца представлена на Рисунке 1.5.1.1.

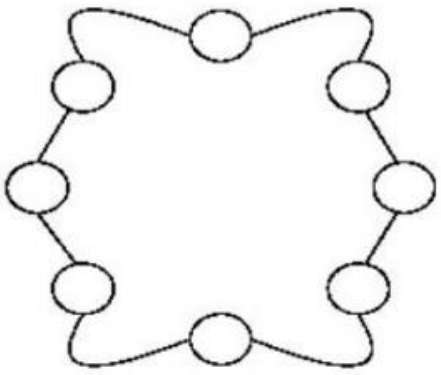

Рисунок 1.5.1.1 – Социальная структура «Кольцо»

1.5.2 Ручной расчёт

Для n = 3 частиц ручной расчёт локального роевого алгоритма с кольцевой структурой социальной сети полностью совпадает с расчётом глобального роевого алгоритма, потому что каждая из частиц может обмениваться с двумя оставшимися соседями (со всеми частицами в рое).

1.6 Программная реализация

Код реализации глобального роевого алгоритма представлен в Приложении А.

Количество частиц, как и число итераций, в алгоритме принято равным ста. Для каждой итерации выводится её номер, точка глобального минимума и значение функции в этой точке. Результат работы глобального роевого алгоритма представлен на Рисунке 1.6.1.

```
----------------------------------------
Итерация: 99/100
Лучшая позиция: [0.007849194860625186, -1.0012552558296983]
Лучшее значение функции: 3.018486793979
----------------------------------------
Итерация: 100/100
Лучшая позиция: [0.007849194860625186, -1.0012552558296983]
Лучшее значение функции: 3.018486793979
----------------------------------------
Лучшая позиция: [0.007849194860625186, -1.0012552558296983]
Лучшее значение функции: 3.018486793979
PS C:\python_projects>
```

Рисунок 1.6.1 - Результат работы глобального роевого алгоритма

Код реализации локального роевого алгоритма с кольцевой социальной сетью представлен в Приложении Б.

Количество частиц и число итераций совпадают с параметрами для глобального алгоритма роя частиц.

```
PS C:\python_projects> & C:/Users/Влад/AppData/Local/Programs/Python/Python312/python.exe "c:/python_projects/Local Best PSO.py"
Лучшая позиция: [-0.008192219040955018, -1.0089374014914096]
Лучшее значение функции: 3.035971410552
PS C:\python_projects>
```

Рисунок 1.6.2 - Результат работы локального роевого алгоритма

ЗАКЛЮЧЕНИЕ

В ходе выполнения данной работы был изучен алгоритм роя частиц, проведён его ручной расчёт для задачи поиска глобального минимума функции, а также разработаны программы на языке Python для минимизации функции Гольдштейна-Прайса с использованием глобального и локального роевого алгоритма с топологией «кольцо».

Основное преимущество роевого алгоритма заключается в его гибкости и способности избегать локальных минимумов за счёт коллективного поиска, что делает его полезным инструментом в численной и комбинаторной оптимизации. Рассмотренные версии глобального и локального роевого алгоритма имеют общую социальную компоненту, направляющую частицы к лучшей позиции, но отличаются свойствами сходимости. Глобальный подход обеспечивает быструю сходимость за счёт тесного взаимодействия частиц, однако это сужает пространство поиска, увеличивая риск преждевременной сходимости к локальным экстремумам. В отличие от него, локальный роевый алгоритм обладает большим разнообразием возможных решений, что снижает вероятность преждевременной сходимости и способствует исследованию более широкого пространства решений.

.

СПИСОК ИНФОРМАЦИОННЫХ ИСТОЧНИКОВ

1. Сорокин, А. Б. Введение в роевой интеллект: теория, расчеты и приложения [Электронный ресурс] : Учебно-методическое пособие / А. Б. Сорокин – Москва: Московский технологический университет (МИРЭА), 2019.
2. Сорокин, А. Б. Безусловная оптимизация. [Электронный ресурс] : учебно-метод. пособие / А. Б. Сорокин, О. В. Платонова, Л. М. Железняк — М. РТУ МИРЭА , 2020.
3. Сорокин, А. Б. Введение в генетические алгоритмы: теория, расчеты и приложения. [Электронный ресурс] : учебно-метод. пособие / А. Б. Сорокин — М. МИРЭА , 2018.
4. Казакова, Е. М. Краткий обзор методов оптимизации на основе роя частиц // Вест. КРАУНЦ. Физ.-мат. науки. 2022. №2. URL: https://cyberleninka.ru/article/n/kratkiy-obzor-metodov-optimizatsii-na-osnove-roya-chastits (Дата обращения: 11.11.2024).

ПРИЛОЖЕНИЯ

Приложение А — Код реализации глобального роевого алгоритма.

Приложение Б — Код реализации локального роевого алгоритма.

Приложение А

Код реализации глобального роевого алгоритма

Листинг А – Реализация глобального роевого алгоритма

```
import random
from typing import List, Tuple, Callable

def goldstein_price(x: float, y: float) -> float:
    '''Функция Голдштейна-Прайса для оптимизации.'''
    term1 = (1 + (x + y + 1)**2 * (19 - 14 * x + 3 *
             x**2 - 14 * y + 6 * x * y + 3 * y**2))
    term2 = (30 + (2 * x - 3 * y)**2 * (18 - 32 * x +
             12 * x**2 + 48 * y - 36 * x * y + 27 * y**2))
    return term1 * term2

class Particle:
    def __init__(self, bounds: List[Tuple[float, float]], fitness_function:
Callable[..., float]):
        '''
        Инициализирует частицу с случайными позицией и скоростью.
        Параметры:
            bounds (List[Tuple[float, float]]): Ограничения для координат каждой
частицы.
            fitness_function (Callable[..., float]): Целевая функция для
оптимизации.
        '''
        self.position = [random.uniform(*bound) for bound in bounds]
        self.velocity = [random.uniform(-1, 1) for _ in bounds]
        self.best_position = self.position[:]
        self.fitness_function = fitness_function
        self.best_value = self.fitness_function(*self.position)

    def __str__(self) -> str:
        return f"(Координаты: {[f'{pos:.4f}' for pos in self.position]}; " + \
            f"Скорость: {[f'{vel:.4f}' for vel in self.velocity]}; "  + \
                f"Лучшие координаты: {[f'{b_pos:.4f}' for b_pos in
self.best_position]}; " + \
                    f"Лучшее значение функции: {self.best_value:.4f})."

    def update_velocity(self, global_best_position: List[float], c1: float =
2.0, c2: float = 2.0) -> None:
        '''
        Обновляет скорость частицы на основе её лучшей позиции и глобальной
лучшей позиции роя.
        Параметры:
            global_best_position (List[float]): Лучшая позиция в рое.
            c1 (float): Коэффициент когнитивного компонента.
            c2 (float): Коэффициент социального компонента.
        '''
        for i in range(len(self.position)):
            r1, r2 = random.random(), random.random()
            cognitive = c1 * r1 * (self.best_position[i] - self.position[i])
            social = c2 * r2 * (global_best_position[i] - self.position[i])
            self.velocity[i] += cognitive + social

    def update_position(self, bounds: List[Tuple[float, float]]) -> None:
        '''
        Обновляет позицию частицы с учётом ограничений и обновляет её лучшую
позицию.
        Параметры:
```

Окончание Листинга А

```
            bounds (List[Tuple[float, float]]): Ограничения для координат.
        '''
        for i in range(len(self.position)):
            self.position[i] += self.velocity[i]
            self.position[i] = max(
                min(self.position[i], bounds[i][1]), bounds[i][0])
        current_value = self.fitness_function(*self.position)
        if current_value < self.best_value:
            self.best_position = self.position[:]
            self.best_value = current_value

class Swarm:
    def __init__(self, fitness_function: Callable[..., float], bounds:
List[Tuple[float, float]],
                 num_particles: int, max_iterations: int):
        '''
        Инициализирует рой частиц для оптимизации.
        Параметры:
            fitness_function (Callable[..., float]): Целевая функция для
оптимизации.
            bounds (List[Tuple[float, float]]): Ограничения для координат.
            num_particles (int): Количество частиц в рое.
            max_iterations (int): Максимальное количество итераций для
оптимизации.
        '''
        self.particles = [Particle(bounds, fitness_function)
                          for _ in range(num_particles)]
        self.global_best_position = min(
            self.particles, key=lambda p: p.best_value).best_position[:]
        self.global_best_value = fitness_function(*self.global_best_position)
        self.max_iterations = max_iterations

    def optimize(self) -> Tuple[List[float], float]:
        '''
        Выполняет оптимизацию, обновляя позиции и скорости частиц.
        Возвращает:
            Tuple[List[float], float]: Лучшая позиция и значение целевой
функции.
        '''
        for k in range(self.max_iterations):
            print(f"Итерация: {k + 1}/{self.max_iterations}")
            for particle in self.particles:
                particle.update_position(bounds)
                if particle.best_value < self.global_best_value:
                    self.global_best_position = particle.best_position[:]
                    self.global_best_value = particle.best_value
            for particle in self.particles:
                particle.update_velocity(self.global_best_position)
            print(f"Лучшая позиция: {self.global_best_position}")
            print(f"Лучшее значение функции: {self.global_best_value:.12f}")
            print('-' * 40)
        return self.global_best_position, self.global_best_value

bounds = [(-2, 2), (-2, 2)]
num_particles = 100  # Количество частиц
max_iterations = 100  # Максимальное количество итераций
swarm = Swarm(goldstein_price, bounds, num_particles, max_iterations)
best_position, best_value = swarm.optimize()
print("Лучшая позиция:", best_position)
print(f"Лучшее значение функции: {best_value:.12f}")
```

Приложение Б

Код реализации локального роевого алгоритма

Листинг Б – Реализация локального роевого алгоритма

```
import random
from typing import List, Tuple, Callable

def goldstein_price(x: float, y: float) -> float:
    '''Функция Голдштейна-Прайса для оптимизации.'''
    term1 = (1 + (x + y + 1)**2 * (19 - 14 * x + 3 * x**2 - 14 * y + 6 * x * y +
3 * y**2))
    term2 = (30 + (2 * x - 3 * y)**2 * (18 - 32 * x + 12 * x**2 + 48 * y - 36 *
x * y + 27 * y**2))
    return term1 * term2

class Particle:
    def __init__(self, bounds: List[Tuple[float, float]], fitness_function:
Callable[..., float]):
        '''
        Инициализирует частицу с случайными позицией и скоростью.
        Параметры:
            bounds (List[Tuple[float, float]]): Ограничения для координат каждой
частицы.
            fitness_function (Callable[..., float]): Целевая функция для
оптимизации.
        '''
        self.position = [random.uniform(*bound) for bound in bounds]
        self.velocity = [random.uniform(-1, 1) for _ in bounds]
        self.best_position = self.position[:]
        self.fitness_function = fitness_function
        self.best_value = self.fitness_function(*self.position)

    def __str__(self) -> str:
        '''Возвращает строковое представление текущего состояния частицы.'''
        return f"(Координаты: {[f'{pos:.4f}' for pos in self.position]}; " + \
               f"Скорость: {[f'{vel:.4f}' for vel in self.velocity]}; "  + \
               f"Лучшие координаты: {[f'{b_pos:.4f}' for b_pos in
self.best_position]}; " + \
               f"Лучшее значение функции: {self.best_value:.4f})."

    def update_velocity(self, local_best_position: List[float], c1: float = 2.0,
c2: float = 2.0) -> None:
        '''
        Обновляет скорость частицы на основе её лучшей позиции и локальной
лучшей позиции.
        Параметры:
            local_best_position (List[float]): Локальная лучшая позиция.
            c1 (float): Коэффициент когнитивного компонента.
            c2 (float): Коэффициент социального компонента.
        '''
        for i in range(len(self.position)):
            r1, r2 = random.random(), random.random()
            cognitive = c1 * r1 * (self.best_position[i] - self.position[i])
            social = c2 * r2 * (local_best_position[i] - self.position[i])
            self.velocity[i] += cognitive + social

    def update_position(self, bounds: List[Tuple[float, float]]) -> None:
        '''
        Обновляет позицию частицы с учётом ограничений и обновляет её лучшую
позицию.
```

Продолжение Листинга Б

```
        Параметры:
            bounds (List[Tuple[float, float]]): Ограничения для координат.
        '''
        for i in range(len(self.position)):
            self.position[i] += self.velocity[i]
            self.position[i] = max(min(self.position[i], bounds[i][1]),
bounds[i][0])
        current_value = self.fitness_function(*self.position)
        if current_value < self.best_value:
            self.best_position = self.position[:]
            self.best_value = current_value

class Swarm:
    def __init__(self, fitness_function: Callable[..., float], bounds:
List[Tuple[float, float]],
                 num_particles: int, max_iterations: int):
        '''
        Инициализирует рой частиц для оптимизации.
        Параметры:
            fitness_function (Callable[..., float]): Целевая функция для
оптимизации.
            bounds (List[Tuple[float, float]]): Ограничения для координат.
            num_particles (int): Количество частиц в рое.
            max_iterations (int): Максимальное количество итераций для
оптимизации.
        '''
        self.particles = [Particle(bounds, fitness_function) for _ in
range(num_particles)]
        self.fitness_function = fitness_function
        self.max_iterations = max_iterations

    def get_local_best(self, index: int) -> List[float]:
        '''
        Находит лучшую позицию среди соседей данной частицы.
        Параметры:
            index (int): Индекс текущей частицы.
        Возвращает:
            List[float]: Лучшая позиция среди соседей.
        '''
        neighbors_indices = [(index - 1) % len(self.particles), index, (index +
1) % len(self.particles)]
        neighbors = [self.particles[i] for i in neighbors_indices]
        best_neighbor = min(neighbors, key=lambda p: p.best_value)
        return best_neighbor.best_position

    def optimize(self) -> Tuple[List[float], float]:
        '''
        Выполняет оптимизацию, обновляя позиции и скорости частиц.
        Возвращает:
            Tuple[List[float], float]: Лучшая позиция и значение целевой
функции.
        '''
        for k in range(self.max_iterations):
            for i, particle in enumerate(self.particles):
                local_best_position = self.get_local_best(i)
                particle.update_velocity(local_best_position)
            for particle in self.particles:
                particle.update_position(bounds)
        best_particle = min(self.particles, key=lambda p: p.best_value)
        return best_particle.best_position, best_particle.best_value
```

Окончание Листинга Б

```
bounds = [(-2, 2), (-2, 2)]
num_particles = 100  # Количество частиц
max_iterations = 100  # Максимальное количество итераций
swarm = Swarm(goldstein_price, bounds, num_particles, max_iterations)
best_position, best_value = swarm.optimize()
print("Лучшая позиция:", best_position)
print(f"Лучшее значение функции: {best_value:.12f}")
```