

Оглавление

ВВЕДЕНИЕ.....	4
1. АЛГОРИТМ РОЯ ЧАСТИЦ.....	5
1.1. Естественная мотивация.....	5
1.2. Описание роевого алгоритма	6
1.3. Вариации роевого алгоритма	9
1.4. Основные аспекты и параметры роевых алгоритмов.....	10
1.5. Реализация классов программного кода	14
1.6. Пример работы программного кода алгоритма	23
2. АЛГОРИТМ МУРАВЬИНОЙ КОЛОНИИ.....	26
2.1. Естественная мотивация.....	26
2.2. Описание муравьиного алгоритма	28
2.3. Пример расчета итерации.....	31
2.4. Программная реализация	32
2.5. Обзор модификаций муравьиного алгоритма	46
3. АЛГОРИТМ ПЧЕЛИНОЙ КОЛОНИИ.....	49
3.1. Естественная мотивация.....	49
3.2. Описание пчелиного алгоритма.....	50
3.3. Пример расчета итерации.....	54
3.4. Реализация классов программного кода.....	55
3.5. Пример работы программного кода алгоритма	66
4. ОПИСАНИЕ МАЛОИЗВЕСТНЫХ РОЕВЫХ АЛГОРИТМОВ	73
4.1. Алгоритм роя светлячков	73
4.2. Алгоритм кукушкиного поиска	76
4.3. Алгоритм летучих мышей.....	79
4.4. Алгоритм обезьяньего поиска.....	82
4.5. Алгоритм прыгающих лягушек	85
4.6. Алгоритм поиска косяком рыб	86
4.7. Сорняковый алгоритм.....	89
4.8. Бактериальная оптимизация	91
4.9. Алгоритм гравитационного поиска.....	97
Контрольные вопросы	100
Список литературы	101

ВВЕДЕНИЕ

Слаженное поведение многих животных, насекомых и бактерий всегда вызывало интерес человека. Такое коллективное поведение описывает децентрализованную самоорганизующуюся систему, так называемый «роевой интеллект». Данный термин был введен Херардо Бени и Ван Цзином в 1989 году, в контексте системы клеточных роботов. Роевой интеллект относится к теории искусственного интеллекта как метод нахождения оптимальных решений [1].

Роевые алгоритмы, также, как и эволюционные, используют популяцию особей – потенциальных решений проблемы и метод стохастической оптимизации, который навеян (моделирует) социальным поведением птиц или рыб в стае или насекомых в рое. Аналогично эволюционным алгоритмам начальная популяция потенциальных решений также генерируется случайным образом и далее ищется оптимальное решение проблемы в процессе выполнения роевого алгоритма. Первоначально в роевом алгоритме была предпринята попытка моделировать поведение стаи птиц, которая обладает способностью порой внезапно и синхронно перегруппироваться и изменять направление полета при выполнении некоторой задачи. В отличие от генетического алгоритма здесь не используются генетические операторы, в роевом алгоритме особи (называемые частицами) летают в процессе поиска в гиперпространстве поиска решений и учитывают успехи своих соседей. Если одна частица видит хороший (перспективный) путь (в поисках пищи или защиты от хищников), то остальные частицы способны быстро последовать за ней, даже если они находились в другом конце роя. С другой стороны, в рое, для сохранения достаточно большого пространства поиска, должны быть частицы с долей случайности в своем поведении (движении).

В учебно-методическом пособии рассмотрены наиболее популярные алгоритмы роевого алгоритма. Изложены теоретические основы разработки роевых алгоритмов, описаны процедуры математического поиска, в которых комментируется решение практических задач, алгоритмы рассмотрены с момента постановки задачи до выполнения процедур работы. Приведены конкретные примеры решения задач оптимизации, рассмотрены реализации программных кодов роевых алгоритмов.

Пособие предназначено для студентов 3-го курса квалификации бакалавр, обучающихся по направлению 09.03.04 «Программная инженерия» по профилю «Системы поддержки принятия решений».

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.

Самая первая компьютерная модель роя частиц была придумана ещё в далёком 1986 Крейгом Рейнольдсом. Он, занимаясь созданием графической модели, придумал довольно простые правила поведения для частиц роя, действуя по которым, рой выглядел крайне похожим на реальный аналог птичьего роя. Классическая модель роя частиц была создана лишь в 1995 году Расселом Эберхартом и Джеймсом Кеннеди. Их модель отличается тем, что частицы-агенты роя, помимо подчинения неким правилам обмениваются информацией друг с другом, а текущее состояние каждой частицы характеризуется местоположением частицы в пространстве решений и скоростью перемещения [2].

1.1. Естественная мотивация

Работы в области роевого интеллекта были инспирированы исследованиями разнообразных реальных колоний. Например, метод роя частиц основан на исследовании и моделировании коллективного поведения стай птиц, проведенного Крейгом Рейнольдсом. Целью этих исследований было построение реалистической анимации полета птиц. Особенностью поведения птичьих стай является то, что в них нет никакого единого центра управления, тем не менее вся стая демонстрирует весьма цельное поведение. Рейнольдсом были сформулированы три простых принципа, которых должна придерживаться каждая отдельная птица (рис.1.1) [3]:

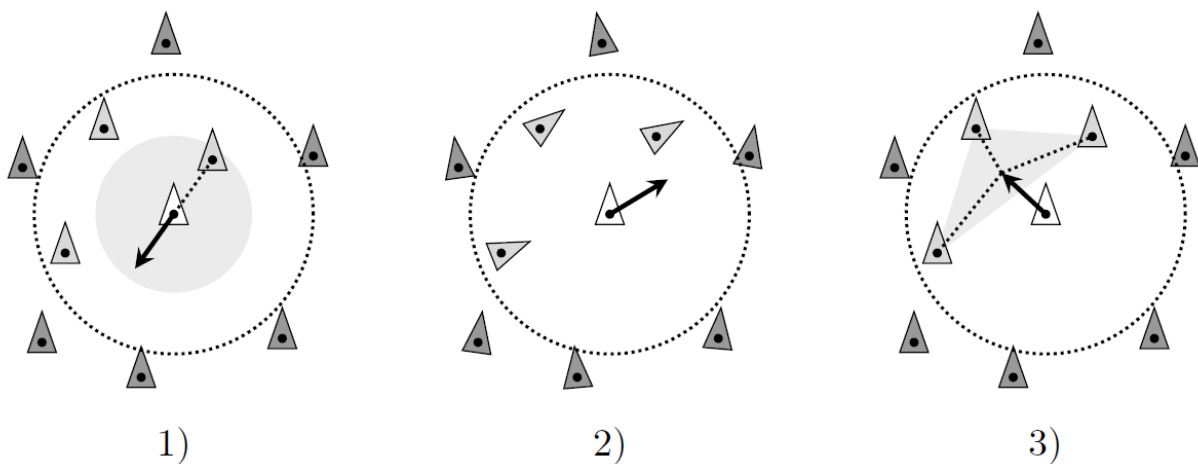


Рис. 1.1. Правила поведения птиц в модели Рейнольдса

1) каждая птица старается не приближаться к другим птицам (или другим объектам) на расстояние меньше некоторой заданной величины;

2) каждая птица старается выбрать свой вектор скорости наиболее близким к среднему вектору скорости среди всех птиц в своей локальной окрестности;

3) каждая птица старается расположиться в геометрическом центре масс своей локальной окрестности.

Первый принцип предназначен для того, чтобы избежать столкновений среди птиц, второй — координирует скорость и направление полета, третий — заставляет каждую птицу не отрываться от стаи (а в идеале — расположиться внутри стаи). Иногда эти принципы входят в противоречие друг с другом, в этих случаях целесообразно ввести приоритет отдельных правил и следовать правилам с наивысшим приоритетом.

Моделирование, основанное на перечисленных принципах, показывает весьма реалистическое поведение стай, способных, в частности, разделяться на несколько групп при встрече с препятствиями (домами, деревьями) и затем, спустя некоторое время, вновь соединяться в общую стаю.

Переход от моделирования коллективного поведения к коллективной оптимизации основан на следующей биологической идее: организмы объединяются в колонии для улучшения своих условий существования — каждый организм в колонии в среднем имеет больше шансов на выживание в борьбе с хищниками, колония может более эффективно производить поиск, обработку и хранение пищи по сравнению с отдельными особями и т. д. Другими словами любая колония организмов в течение всего времени своего существования с той или иной степенью эффективности решает различные оптимизационные задачи, чаще всего — многокритериальные (например, максимизация количества пищи с одновременной минимизацией потерь от хищников) [4].

1.2. Описание роевого алгоритма

Модель, положенная в основу этого метода роевого алгоритма, была получена упрощением модели Рейнолдса, в результате чего отдельные особи популяции стали представляться отдельными объектами, не имеющими размера, но обладающими некоторой скоростью. В силу крайней схожести с материальными частицами получившиеся простые объекты стали называться частицами, а их популяция — роем. Метод роя частиц предназначен для решения задач многомерной непрерывной оптимизации и основан на моделировании социального поведения колоний животных, выполняющих коллективный поиск мест с наилучшими условиями существования.

Пусть задана функция $f: R^n \rightarrow R$, требуется найти глобальный максимум этой функции, т. е. точку x_0 такую, что $f(x_0) \geq f(x)$ для любого $x \in R^n$.

Суть подхода, на котором основан метод роя частиц, заключается в том, что глобальный максимум функции f ищется с помощью системы (роя), состоящей из m частиц. Частицы выполняют поиск, перемещаясь по пространству решений R^n . Положение i -ой частицы задается вектором $x_i \in R^n$, значение $f(x_i)$ определяет функцию качества этой частицы в текущий момент времени.

Каждая частица в рое обладает своей собственной скоростью $v_i \in R^n$, которая определяет как изменяются координаты частицы со временем:

$$x_i \leftarrow x_i + \tau v_i,$$

где τ — некоторая единица измерения скорости (продолжительность одного такта работы алгоритма, например, можно положить $\tau = 1$). Последовательность (дискретная) положений i -ой частицы называется ее траекторией.

Ключевая особенность метода роя частиц заключается в способе обновления скорости отдельных частиц, которое выполняется по формуле

$$v_i \leftarrow v_i + \alpha(p_i - x_i) + \beta(g - x_i).$$

Первое слагаемое в. представляет собой инерцию частицы. Вектор p_i (второе слагаемое) служит простейшей моделью индивидуальной памяти — он равен лучшей точке траектории i -ой частицы за все время ее существования. Говорят, что второе слагаемое реализует принцип простой ностальгии — каждая частица «хочет» вернуться в ту точку, где ею было достигнуто лучшее значение функции f . Вектор g (третье слагаемое), представляет собой лучшую точку, обнаруженной за время своего существования всем роем, т. е. представляет собою некую коллективную память. Третье слагаемое определяет некоторую простую схему социального взаимодействия между отдельными частицами роя.

Другими словами, изменение скорости каждой частицы (т. е. ее ускорение) определяется как некая взвешенная сумма двух векторов, первый из которых направлен на лучшую точку, обнаруженную данной частицей, а второй — на лучшую точку, обнаруженную всем роем. Коэффициенты α и β могут выбираться из разных соображений. Численные эксперименты показали, что лучшей является вероятностная схема — либо оба коэффициента выбираются случайным образом из диапазона $[0, 1]$, либо значение α выбирается случайным образом из этого диапазона, а значение β полагается равным $1 - \alpha$.

Тогда метод роя частиц формально может быть описан следующим образом [3]:

- 1: Случайное распределение m частиц по пространству решений R^n ;
- 2: Вычисление векторов p_i и g ;
- 3: Случайная инициализация скоростей частиц;

```

4: while не выполнен критерий останова do;
5: for  $i \in [1 \dots m]$  do;
6:  $\alpha, \beta \leftarrow \text{Random}(0, 1)$ 
7:  $v_i \leftarrow v_i + \alpha(p_i - x_i) + \beta(g - x_i)$ 
8:  $x_i \leftarrow x_i + \tau v_i$ 
9: if  $f(x_i) > f(p_i)$  then
10:  $p_i \leftarrow x_i$ 
11: if  $f(x_i) > f(g)$  then
12:  $g \leftarrow x_i$ 
13: return  $g$ 

```

Как видно, алгоритм роя частиц – итеративный процесс, постоянно находящийся в изменении (рис. 1.2).

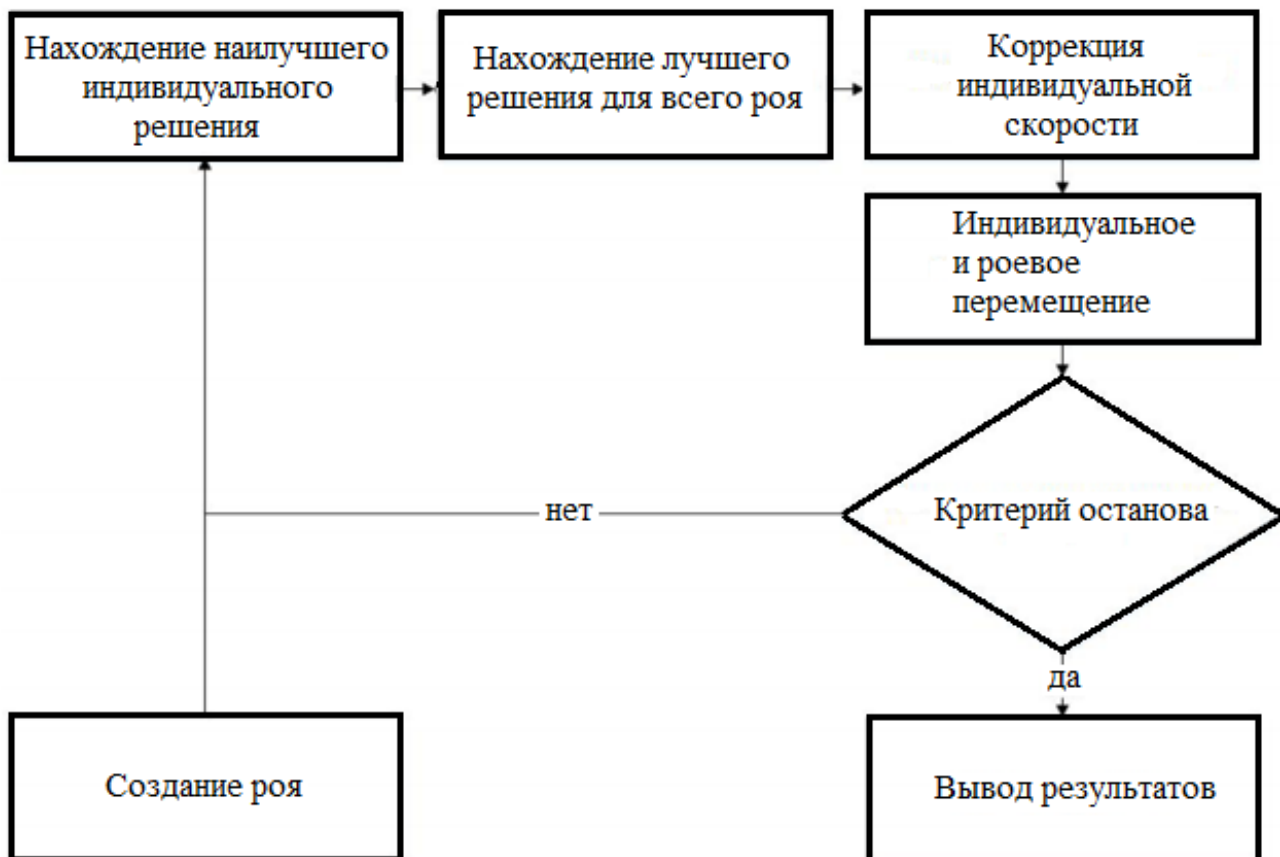


Рис. 1.1. Алгоритм работы метода роя частиц

Метод роя частиц является, по сути, метаэвристикой, хорошо зарекомендовавшей себя при решении различных оптимизационных задач. Его отличительной особенностью от многих других методов является то, что для метода роя частиц необходимо уметь вычислять только значение оптимизируемой функции, но не ее градиент. Т. е. функция, подлежащая оптимизации, не обязана быть дифференцируемой, более того, она может быть разрывной, зашумленной и т. п. С другой стороны, в силу того, что метод оперирует понятием скоро-

сти частиц, необходимым условием его применимости является непрерывность области определения функции. В частности, это означает, что данный метод неприменим напрямую к задачам дискретной оптимизации.

1.3. Вариации роевого алгоритма

Большая часть предложенных вариаций базового алгоритма Кеннеди и Эберхарта касается модификации формулы скорости. Основная проблема, связанная с применением этой формулы, заключается в том, что возникающие при этом скорости частиц мог быть сколь угодно большими, это приводит некоторую неустойчивость в работу метода. Простейший способ ограничить скорости частиц заключается в прямом запрете на превышение некоторой максимальной скорости v_{max} . Другой, более естественный, способ ограничения модуля скорости заключается в введении некоторого «сопротивления среды», задаваемого положительным коэффициентом $\gamma < 1$ [3]:

$$v_i \leftarrow \gamma(v_i + \alpha(p_i - x_i) + \beta(g - x_i))$$

Другие, менее значительные, изменения базового алгоритма касаются следующих моментов [5]:

- способ начального распределения частиц по пространству
- решений;
- способ выбора начальной скорости частиц;
- схема вычисления коэффициентов α и β ;
- обновление вектора g только после вычисления новых координат всеми частицами роя;
- использование в формуле скорости вектора g , вычисленного не для всего роя (глобальная операция), а только для некоторого подмножества i -ой частицы (ее локальной окрестности в некоторой метрике, не связанной с пространством решений).

Также были предложены и исследованы вариации метода роя частиц, применимые к задачам дискретной оптимизации. Проблемой является то, что в дискретном пространстве решений (например, на множестве двоичных последовательностей) понятие скорости частицы полностью теряет свой смысл. Один из вариантов решения этой проблемы, предложенный самим Эберхартом, заключается в том, что k -ая компонента скорости частицы, ограниченная интервалом $[0, 1]$ трактуется, как вероятность изменения k -ой координаты этой частицы — чем больше скорость, тем чаще меняется координата (например, 0 на 1 и наоборот). Такой прием позволяет отделить пространство решений от пространства скоростей и, в частности, эффективно решать задачи оптимизации в дискретных пространствах решений.

Также разработаны две базовые схемы распараллеливания метода роя частиц — с централизованным и локальным управлением. В любом случае частицы помещаются на различные процессоры вычислительной системы, либо по одной (мелкозернистый параллелизм), либо группами (крупнозернистый параллелизм). В первом случае используется вариант метода с глобальным вычислением вектора g . Каждый рабочий процесс независимо от остальных вычисляет координаты x_i , вектор скорости v_i и вектор p_i , после чего обменивается информацией (p_i посылается мастер-процессу, g — рабочему процессу). При увеличении числа рабочих процессов будет возрастать и время ожидания рабочих процессов, т. е. снижаться общая эффективность. Снизить относительное время ожидания можно, если разрешить каждой частице делать несколько перемещений между двумя последующими стадиями синхронизации [5].

В варианте с локальным управлением мастер-процесс вообще не используется. Вместо этого вводится понятие локальной окрестности частицы, под которой понимается некоторое подмножество частиц, распределенных на процессоры, соседние с данным. Работа параллельного алгоритма производится по той же схеме, что и для первого варианта, с тем отличием, что вектор g вычисляется теперь не глобально, а локально внутри локальной окрестности каждой частицы: каждый процесс опрашивает всех своих соседей и выбирает лучший из имеющихся у этих процессов векторов g . Этот вариант является более предпочтительным при реализации на массивно-параллельных вычислительных системах, т. к. он хорошо и легко масштабируется.

1.4. Основные аспекты и параметры роевых алгоритмов

Рассмотрим некоторые аспекты представленных выше роевых алгоритмов, к которым относятся вопросы инициализации, условий останова, вычисления фитнес-функций и т.п. Как было показано ранее, процесс поиска решения в роевых алгоритмах является итеративным, который продолжается пока не выполнено условие останова. Одна итерация содержит все шаги основного цикла, включая определение персональных и глобальных лучших позиций и коррекцию скорости каждой частицы. В каждой итерации производится оценка значений фитнес-функций частиц. Оценка качества потенциального решения выполняется на основе вычисления значения фитнес-функции, которая характеризует данную задачу оптимизации. В основном роевой алгоритм выполняется n_s вычислений фитнес-функции за итерацию, где n_s число частиц в рое.

На первом этапе алгоритма производится инициализация роя и управляющих параметров алгоритма. При этом определяются начальные значения скоростей, позиций и персональных лучших позиций частиц, и т.п.

Обычно позиции частиц инициализируются таким образом, чтобы пространство поиска покрывалось равномерно. Следует отметить, что эффективность роевых алгоритмов существенно зависит от начального разнообразия множества потенциальных решений, то есть от того, как первоначально покрыто пространство поиска и как распределены частицы в нем. Если некоторые области пространства поиска не покрыты начальным роем, то алгоритму будет трудно найти оптимум в том случае, когда он расположен в не охваченном участке. В этом случае алгоритм может найти такой оптимум благодаря моменту частицы, который может направить ее в неисследованную область.

Предположим, что оптимум расположен внутри области, определяемой двумя векторами x_{min} , x_{max} , которые представляют минимальные и максимальные значения по каждой координате. Тогда эффективным методом инициализации начальной позиции частиц является:

$$x(0) = x_{min,j} + r_j(x_{max,j} - x_{min,j}), \forall j = 1, \dots, n_x, \forall i = 1, \dots, n_x$$

где $r_j \sim U(0,1)$

Начальные скорости частиц при этом можно положить нулевыми $v_i(0) = 0$. В другом варианте начальные скорости можно инициализировать случайными значениями из некоторого диапазона, но делать это необходимо осторожно. Случайная инициализация позиций частиц уже определяет начальное движение в случайных направлениях. Если, однако, выполняется случайная инициализация скоростей, то их значения не должны быть слишком большими, чтобы частицы не вышли из «зоны интереса», что может существенно ухудшить сходимость.

Персональная лучшая позиция для каждой частицы определяется позицией частицы в момент $t = 0$, то есть $y_i(0) = x_i(0)$. Различные схемы инициализации позиций частиц исследованы для покрытия пространства поиска: на основе последовательностей. При решении реальных задач важно, прежде всего, чтобы частицы равномерно покрывали пространство поиска.

Следующий аспект касается условия останова процесса поиска решения, где необходимо учитывать следующее [5]:

- 1) критерий не должен вызывать преждевременной сходимости роевого алгоритма в локальных оптимумах;
- 2) критерий должен предотвращать чрезмерно большие вычисления вследствие частой оценки фитнес-функции частиц.

В настоящее время предложен ряд критериев останова, основными из которых являются следующие:

- *Останов по максимальному числу итераций* (при превышении заданного порога). Очевидно, что в случае малого порога числа итераций процесс может остановиться до того, как будет найдено хорошее решение. Этот критерий обычно используется совместно с критерием сходимости. Данный критерий полезен, когда необходимо за ограниченное заданное время найти лучшее решение.

- *Останов по найденному приемлемому решению.* Предположим, что x^* представляет оптимум для целевой функции f . Тогда критерий окончания можно сформулировать в терминах близости найденного лучшего решения x к оптимуму $f(x_i) \leq |f(x^*) - \varepsilon|$, то есть при достижении достаточно малой ошибки ε . Значение порога ошибки ε необходимо выбирать осторожно. Если значение ε слишком велико, то очевидно процесс поиска может остановиться, если найдено не очень хорошее решение. С другой стороны, если значение слишком мало, то процесс может вовсе не сойтись. Это особенно характерно для классического роевого алгоритма. Кроме этого, данный критерий предполагает априорное знание оптимального значения, которое не всегда известно, кроме случаев минимизации ошибки (например, в процессе обучения).

- *Останов по отсутствию улучшения решения за заданное число итераций.* Есть различные способы измерения улучшения получаемых решений. Например, среднее изменение позиции частиц мало, то можно предположить, что процесс поиска сошелся. С другой стороны, если среднее значение скорости частицы за некоторое число итераций примерно равно нулю, то возможны только незначительные изменения позиции частиц и процесс поиска можно остановить. К сожалению, данный критерий требует ввода двух параметров: диапазона изменения итераций и пороговое значение наблюдаемых величин.

- *Останов при стремлении нормализованного радиуса роя к нулю.* Определим нормализованный радиус роя следующим образом:

$$R_{norm} = \frac{R_{max}}{diameter(S)},$$

где $diameter(S)$ - диаметр начального роя и максимальный радиус R_{max} определяется следующим образом

$$R_{max} = ||x_m - \hat{y}||, m = 1, \dots, n_s$$

$$||x_m - \hat{y}|| \geq ||x_i - \hat{y}||, \forall i = 1, \dots, n_s$$

Можно считать, что алгоритм сошелся, если $R_{norm} < \varepsilon$. Если ε слишком велико, то процесс поиска может закончиться раньше, чем будет найдено хо-

рошее решение. С другой стороны при малом значении ε может потребоваться слишком большое число итераций для формирования компактного роя.

- *Останов по малому значению наклона (крутизны) целевой функции.*

Рассмотренные критерии учитывают только относительное расположение частиц в пространстве поиска и не принимают во внимание информацию о крутизне целевой функции. Для учета изменений целевой функции часто используют следующее отношение:

$$f'(t) = \frac{f(\hat{y}(t)) - f(\hat{y}(t-1))}{f(\hat{y}(t))}$$

Если $f'(t) < \varepsilon$ для определенного числа последовательных итераций, то можно считать, что рой сошелся. Этот критерий сходимости, по мнению многих специалистов, превосходит приведенные ранее, так как он учитывает динамические характеристики роя. Однако использование крутизны целевой функции в качестве критерия останова может привести к преждевременной сходимости в локальном экстремуме. Поэтому целесообразно использовать данный критерий в сочетании с приведенным ранее критерием нормализованного радиуса роя.

Следует отметить, что сходимость по приведенным критериям, в общем случае, может не соответствовать достижению оптимума (глобального или локального). В данном случае сходимость означает, что рой достиг состояния равновесия, когда частицы стремятся к некоторой точке в пространстве поиска (в общем случае необязательно точке оптимума).

Эффективность роевого алгоритма зависит от ряда параметров, к которым относятся: размерность задачи, число частиц, коэффициенты ускорения, вес инерции, тип и размер соседнего окружения, число итераций. В случае наложения ограничений на возможные скорости частиц необходимо также определить максимальные значения и некоторые коэффициенты. [6]

Рассмотрим основные параметры роевого алгоритма [7].

Размер роя, число частиц n_s , играет большую роль: чем больше частиц, тем больше разнообразие потенциальных решений (при хорошей схеме инициализации, обеспечивающей однородное распределение частиц). Большое число частиц позволяет покрыть большую часть пространства поиска за итерацию. С другой стороны, большое число частиц повышает вычислительную сложность итерации и при этом алгоритм может вырождаться в случайный параллельный поиск. Хотя бывают случаи, что большее число частиц ведет к уменьшению числа итераций при поиске хороших решений. Экспериментально показано, что роевые алгоритмы способны находить оптимальное решение с малым размером

роя от 10 до 30 частиц. В общем случае оптимальный размер роя зависит от решаемой задачи и определяется экспериментально.

Размер соседнего окружения определяет степень влияния социальной компоненты в роевых алгоритмах. Чем меньше соседей у частицы, тем меньше ее взаимодействие с окружением. Малый размер определяет малую скорость сходимости, но дает большую надежность поиска оптимума. Чем меньше размер окружения, тем меньше чувствительность к локальным экстремумам. Для использования преимуществ малого и большого размера окружения часто стартуют с малым размером соседней окрестности и далее в процессе поиска размер окружения увеличивают пропорционально числу выполненных итераций. Такой подход обеспечивает хорошее начальное разнообразие и быструю сходимость частиц к перспективной области поиска.

Число итераций, обеспечивающее нахождение хорошего решения, зависит от решаемой задачи. При малом числе процесс поиска может не успеть сойтись. С другой стороны, большое число итераций, естественно, повышает вычислительную сложность.

1.5. Реализация классов программного кода

Алгоритм метода роя частиц реализовывался на языке программирования Python. Реализация алгоритма роя частиц содержится в пакете `particleswarm`, который написан таким образом, чтобы пользователю этого пакета нужно было бы только написать минимизируемую функцию и задать границы ее определения, не вдаваясь в подробности работы алгоритма. Разберем структуру пакета `particleswarm` подробнее.

Этот пакет содержит два класса:

- `Particle` описывает поведение частицы на каждом шаге итерации;
- `Swarm` – это абстрактный базовый класс, в котором необходимо определить целевую функцию. Именно с классом, производным от `Swarm` работает пользователь.

Пользователь должен создать класс, производный от `Swarm` (на диаграмме это `ConcreteSwarm`) и определить метод `_finalFunc()`, который должен вернуть значение целевой функции в зависимости от переданного параметра `position` – координат, где нужно рассчитать значение.

Все координаты в алгоритме представляют собой массив чисел, размерность которого равна размерности задачи. Если быть точнее, то используется класс `numpy.array`, чтобы можно было использовать удобные операции с массивами без необходимости организовывать перебор значений списков.

На рис. 1.2 показаны основные элементы классов `Particle` и `Swarm`.

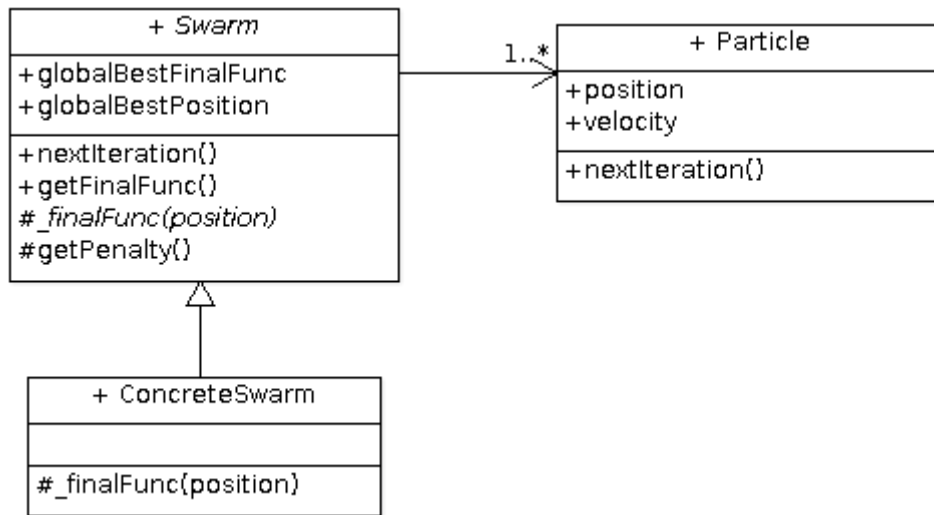


Рис. 1.2. Основные элементы классов Particle и Swarm

Конструктор класса Swarm выглядит следующим образом:

```
def __init__(self,
    swarmsize, # размер роя (количество частиц)
    minvalues, # список, задающий минимальные значения для каждой координаты частицы
    maxvalues, # список, задающий максимальные значения для каждой координаты частицы
    currentVelocityRatio, # общий масштабирующий коэффициент для скорости
    localVelocityRatio, # коэффициент, задающий влияние лучшей точки, найденной каждой частицей, на будущую скорость
    globalVelocityRatio): # коэффициент, задающий влияние лучшей точки, найденной всеми частицами, на будущую скорость
```

Если использовать использовать терминологию, описываемую выше, то `currentVelocityRatio` - это коэффициент k , `localVelocityRatio` - это φp , а `globalVelocityRatio` - это φg . Так как используется каноническая версия алгоритма, то должно выполняться условие

$$localVelocityRatio + globalVelocityRatio > 4.$$

Допустим, что необходимо минимизировать функцию:

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

Класс роя для такой функции будет выглядеть следующим образом (файл `swarm_x2.py`):

```
from particleswarm.swarm import Swarm
class Swarm_X2(Swarm):
```

```

def __init__ (self,
              swarmsize,
              minvalues,
              maxvalues,
              currentVelocityRatio,
              localVelocityRatio,
              globalVelocityRatio):
    Swarm.__init__ (self,
                    swarmsize,
                    minvalues,
                    maxvalues,
                    currentVelocityRatio,
                    localVelocityRatio,
                    globalVelocityRatio)
def _finalFunc (self, position):
    penalty = self._getPenalty (position, 10000.0)
    finalfunc = sum (position * position)

    return finalfunc + penalty

```

В этом коде используется метод `_getPenalty()`, который возвращает штраф, если значение текущих координат частицы (параметр `position`) выходит за пределы области определения функции (как этот предел задается, будет показано ниже). Второй коэффициент (в этом примере 10000,0) определяет, насколько жестким должен быть штраф. Метод `_getPenalty()` для вычисления штрафа выглядит следующим образом:

```

def _getPenalty (self, position, ratio):
    """
    Рассчитать штрафную функцию
    position - координаты, для которых рассчитывается штраф
    ratio - вес штрафа
    """

    penalty1 = sum ([ratio * abs (coord - minval)
                     for coord, minval in zip (position, self.minvalues)
                     if coord < minval ] )

    penalty2 = sum ([ratio * abs (coord - maxval)
                     for coord, maxval in zip (position, self.maxvalues)

```

```
if coord > maxval ] )
```

```
return penalty1 + penalty2
```

При желании можно не использовать метод `_getPenalty()`, а вычислять штраф непосредственно в `_finalFunc()` по другой формуле (например, с использованием экспоненциального нарастания штрафа вместо линейного).

Запуск алгоритма выглядит следующим образом (файл *runoptimize_x2.py*):

```
import numpy
```

```
from swarm_x2 import Swarm_X2
```

```
from utils import printResult
```

```
if __name__ == "__main__":
```

```
    iterCount = 300
```

```
    dimension = 5
```

```
    swarmsize = 200
```

```
    minvalues = numpy.array ([-100] * dimension)
```

```
    maxvalues = numpy.array ([100] * dimension)
```

```
    currentVelocityRatio = 0.1
```

```
    localVelocityRatio = 1.0
```

```
    globalVelocityRatio = 5.0
```

```
    swarm = Swarm_X2 (swarmsize,
```

```
        minvalues,
```

```
        maxvalues,
```

```
        currentVelocityRatio,
```

```
        localVelocityRatio,
```

```
        globalVelocityRatio
```

```
    )
```

```
    for n in range (iterCount):
```

```
        print "Position", swarm[0].position
```

```
        print "Velocity", swarm[0].velocity
```

```
print printResult (swarm, n)
```

```
swarm.nextIteration()
```

Импортируем класс `Swarm_X2`, где описана целевая функция, и из модуля `utils` в функцию `printResult`, которая просто оформляет текущее состояние алгоритма и возвращает результат в виде строки, которую затем выводим в консоль.

Для простоты не используются какие-либо сложные критерии останова функции, а просто алгоритм останавливается на 300-й итерации (переменная `iterCount`).

Наша функция будет 5-мерная (переменная `dimension`), то есть в формуле выше $n = 5$. Количество особей в рое - 200 (переменная `swarmsize`).

Затем определяем интервал, где будем искать минимум. Для этого заполняем массив с минимальными значениями по каждой координате 5-мерного пространства (переменная `minvalues`) и с максимальными (переменная `maxvalues`). В данном случае для всех 5 координат интервал будет $[-100, 100]$, но в общем случае для каждой координаты могут быть свои пределы.

После задаем коэффициенты, о которых говорилось выше (переменные `currentVelocityRatio`, `localVelocityRatio` и `globalVelocityRatio`).

После этого в цикле 300 раз вызываем метод `swarm.nextIteration()`, на каждой итерации выводим текущее состояние роя и, положение и скорость одной из частицы (с номером 0, но она никак не выделяется по сравнению с другими частицами).

Чтобы получить лучшее на данной итерации решение, нужно воспользоваться свойством `globalBestPosition`, который возвращает список (в нашем случае с 5 элементами) со всеми координатами точки, которая в данный момент считается лучшей.

Для того, чтобы узнать значение целевой функции в лучшей на данный момент точке, в классе `Swarm` предусмотрено свойство `globalBestFinalFunc`.

Кроме того, чтобы получить одну частицу (экземпляр класса `Particle`), можно воспользоваться оператором индексации (как в примере, где выводятся координаты и скорость частицы).

Инициализация класса `Swarm` довольно простая, конструктор просто сохраняет переданные параметры, а затем создает нужное количество частиц со случайными значениями координат и скоростей.

```
class Swarm (object):
```



```
"""
```

Базовый класс для роя частиц. Его надо переопределять для конкретной целевой функции

```
"""
```

```
__metaclass__ = ABCMeta
```

```
def __init__(self,
    swarmsize,
    minvalues,
    maxvalues,
    currentVelocityRatio,
    localVelocityRatio,
    globalVelocityRatio):
```

```
"""
```

swarmsize - размер роя (количество частиц)

minvalues - список, задающий минимальные значения для каждой координаты частицы

maxvalues - список, задающий максимальные значения для каждой координаты частицы

currentVelocityRatio - общий масштабирующий коэффициент для скорости

localVelocityRatio - коэффициент, задающий влияние лучшей точки, найденной частицей на будущую скорость

globalVelocityRatio - коэффициент, задающий влияние лучшей точки, найденной всеми частицами на будущую скорость

```
"""
```

```
self.__swarmsize = swarmsize
```

```
assert len (minvalues) == len (maxvalues)
```

```
assert (localVelocityRatio + globalVelocityRatio) > 4
```

```
self.__minvalues = numpy.array (minvalues[:])
```

```
self.__maxvalues = numpy.array (maxvalues[:])
```

```
self.__currentVelocityRatio = currentVelocityRatio
```

```
self.__localVelocityRatio = localVelocityRatio
```

```
self.__globalVelocityRatio = globalVelocityRatio
```

```
self.__globalBestFinalFunc = None
self.__globalBestPosition = None
```

```
self.__swarm = self.__createSwarm ()
```

```
def __createSwarm (self):
```

```
    """
```

```
        Создать рой из частиц со случайными координатами и скоростями
```

```
    """
```

```
        return [Particle (self) for _ in range (self.__swarmsize) ]
```

На каждой итерации активизируется метод nextIteration(), который просто вызывает соответствующий метод у каждой из частицы в рое:

```
def nextIteration (self):
```

```
    """
```

```
        Выполнить следующую итерацию алгоритма
```

```
    """
```

```
    for particle in self.__swarm:
```

```
        particle.nextIteration (self)
```

Теперь настала очередь рассмотреть класс Particle, начнем с конструктора.

```
class Particle (object):
```

```
    """
```

```
        Класс, описывающий одну частицу
```

```
    """
```

```
def __init__ (self, swarm):
```

```
    """
```

```
        swarm - экземпляр класса Swarm, хранящий параметры алгоритма,  
        список частиц и лучшее значение роя в целом
```

```
        position - начальное положение частицы (список)
```

```
    """
```

```
        # Текущее положение частицы
```

```
        self.__currentPosition = self.__getInitPosition (swarm)
```

```
        # Лучшее положение частицы
```

```
        self.__localBestPosition = self.__currentPosition[:]
```

```

        # Лучшее значение целевой функции
        self.__localBestFinalFunc = swarm.getFinalFunc
        (self.__currentPosition)

```

```

        self.__velocity = self.__getInitVelocity (swarm)

```

```

def __getInitPosition (self, swarm):

```

```

    """

```

```

        Возвращает список со случайными координатами для заданного ин-
        тервала изменений

```

```

    """

```

```

    return numpy.random.rand (swarm.dimension) * (swarm.maxvalues -
    swarm.minvalues) + swarm.minvalues

```

```

def __getInitVelocity (self, swarm):

```

```

    """

```

```

        Сгенерировать начальную случайную скорость

```

```

    """

```

```

        assert len (swarm.minvalues) == len (self.__currentPosition)

```

```

        assert len (swarm.maxvalues) == len (self.__currentPosition)

```

```

        minval = -(swarm.maxvalues - swarm.minvalues)

```

```

        maxval = (swarm.maxvalues - swarm.minvalues)

```

```

    return numpy.random.rand (swarm.dimension) * (maxval - minval) + minval

```

Здесь используется функция `numpy.random.rand()`, которая возвращает массив заданного размера из случайных величин в интервале (0, 1).

Конструктор класса `Particle` создает частицу со случайным начальным положением (`self.__currentPosition`), а также со случайной начальной скоростью (`self.__velocity`). Так как частица только создается, то ее начальное положение принимается за лучшее за всю историю похождения частицы (переменная `self.__localBestPosition`), и также сохраняется лучшее за всю историю частицы значение целевой функции (переменная `self.__localBestFinalFunc`)

Целевая функция тоже хранится в классе `Swarm` (точнее, в производном от него классе), поэтому для ее расчета вызывается метод `getFinalFunc()` класса `Swarm`. Обратите внимание, что это не тот метод, который был переопределен в

производном от Swarm классе. Дело в том, что метод getFinalFunc() (публичный, в отличие от переопределенного _finalFunc) не только рассчитывает значение целевой функции в переданной точке, но и смотрит, не стала ли эта точно глобально лучшей.

```
def getFinalFunc (self, position):  
    assert len (position) == len (self.minvalues)  
  
    finalFunc = self._finalFunc (position)  
  
    if (self.__globalBestFinalFunc == None or  
        finalFunc < self.__globalBestFinalFunc):  
        self.__globalBestFinalFunc = finalFunc  
        self.__globalBestPosition = position[:]
```

Именно внутри getFinalFunc() вызывается переопределенная функция.

Создание частиц рассмотрели, вернемся к методу nextIteration() класса Particle. Именно здесь содержится суть алгоритма роя частиц.

```
def nextIteration (self, swarm):  
    # Случайный вектор для коррекции скорости с учетом лучшей позиции  
данной частицы  
    rnd_currentBestPosition = numpy.random.rand (swarm.dimension)  
  
    # Случайный вектор для коррекции скорости с учетом лучшей глобальной  
позиции всех частиц  
    rnd_globalBestPosition = numpy.random.rand (swarm.dimension)  
  
    veloRatio = swarm.localVelocityRatio + swarm.globalVelocityRatio  
    commonRatio = (2.0 * swarm.currentVelocityRatio /  
        (numpy.abs (2.0 - veloRatio - numpy.sqrt (veloRatio ** 2 - 4.0 * veloRa-  
            tio) ) ) )  
  
    # Посчитать новую скорость  
    newVelocity_part1 = commonRatio * self.__velocity  
  
    newVelocity_part2 = (commonRatio *  
        swarm.localVelocityRatio *  
        rnd_currentBestPosition *  
        (self.__localBestPosition - self.__currentPosition) )
```

```

newVelocity_part3 = (commonRatio *
    swarm.globalVelocityRatio *
    rnd_globalBestPosition *
    (swarm.globalBestPosition - self.__currentPosition) )

self.__velocity = newVelocity_part1 + newVelocity_part2 + newVelocity_part3

# Обновить позицию частицы
self.__currentPosition += self.__velocity

finalFunc = swarm.getFinalFunc (self.__currentPosition)
if finalFunc < self.__localBestFinalFunc:
    self.__localBestPosition = self.__currentPosition[:]
    self.__localBestFinalFunc = finalFunc

```

Здесь реализована формула для коррекции скорости, корректируется положение частицы, а затем проверяется, не стала ли новая координата лучшей для данной частицы.

1.6. Пример работы программного кода алгоритма

Для демонстрации работы алгоритма в качестве минимизируемой функции рассмотрим функцию Швепеля:

$$f(X) = \sum_{t=1}^n \left[-x_t \sin(\sqrt{|x_t|}) \right]$$

Ее трехмерный вид показан на рис. 1.3.

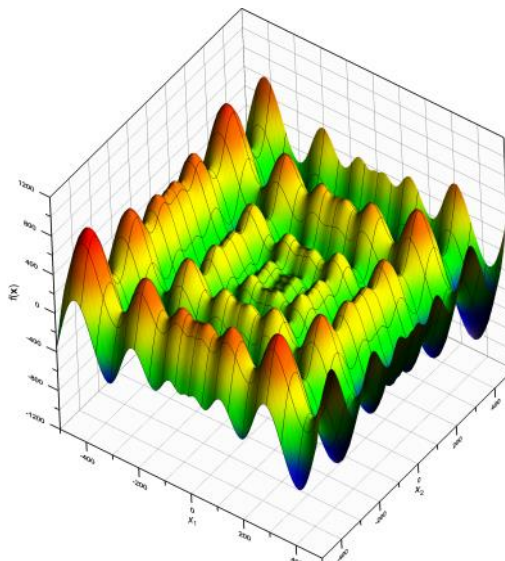


Рис. 1.3. Трехмерный вид функции Швепеля ($n = 2$)

Функция Швепеля для $n=1$ показана на рис. 1.4.

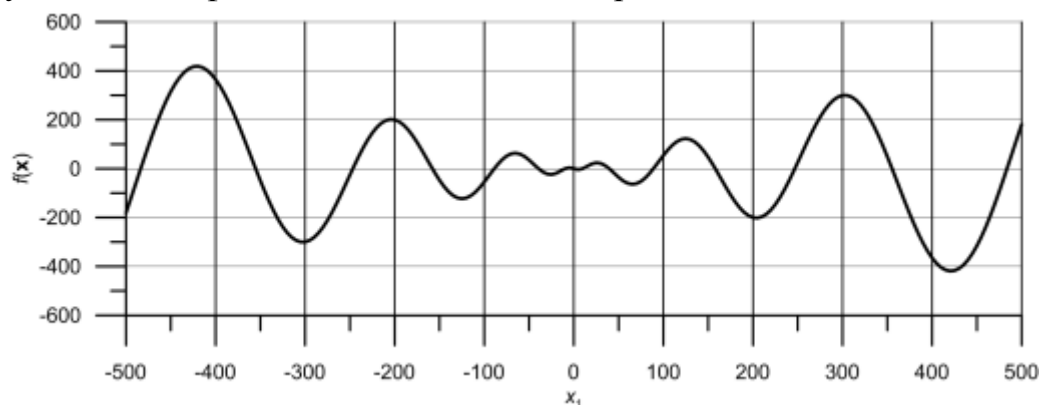


Рис. 1.4. Функция Швепеля для $n=1$

Минимум функции Швепеля на интервале $500 \leq x_i \leq 500$ расположен в точке, где $x_i = 427.9687$ для $i = 1, 2, \dots, n$, а значение функции в этой точке составляет $f(x) = -418.9829n$.

Файл программы функции Швепеля:

```
from particleswarm.swarm import Swarm
import numpy
```

```
class Swarm_Schwefel (Swarm):
```

```
    def __init__ (self,
                  swarmsize,
                  minvalues,
                  maxvalues,
                  currentVelocityRatio,
                  localVelocityRatio,
                  globalVelocityRatio):
```

```
        Swarm.__init__ (self,
                          swarmsize,
                          minvalues,
                          maxvalues,
                          currentVelocityRatio,
                          localVelocityRatio,
                          globalVelocityRatio)
```

```
    def _finalFunc (self, position):
```

```
        function = sum (-position * numpy.sin (numpy.sqrt (numpy.abs (position) ) ) )
        penalty = self._getPenalty (position, 10000.0)
        return function + penalty
```

Вывод работы программы показан на рис. 1.5.

```
Командная строка
Microsoft Windows [Version 10.0.17134.407]
(c) Корпорация Майкрософт (Microsoft Corporation), 2018. Все права защищены.

C:\Users\ido45>python C:\python\swarm_schwefel.py

C:\Users\ido45>x[0]=417.719376749399
x[1]=427.675937840463

C:\Users\ido45>_
```

Рис. 1.5. Результат программы нахождения минимума функции Швевеля

Результат минимума функции оказался достаточно верным. Визуализация минимизации функции Швевеля представлена на рис. 1.6.

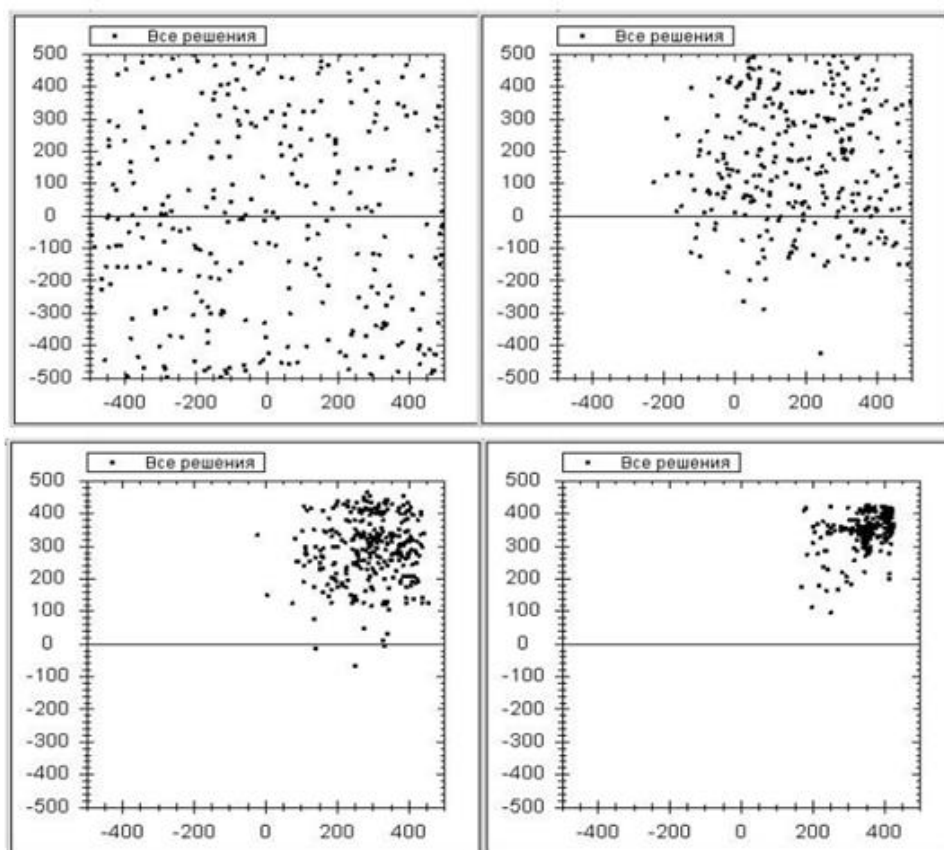


Рис. 1.6. Визуализация минимизации функции Швевеля

В настоящее время роевые алгоритмы применяются при решении задач численной и комбинаторной оптимизации (существует дискретный вариант РА), обучении искусственных нейронных сетей, построении нечетких контроллеров и т.д.

2. АЛГОРИТМ МУРАВЬИНОЙ КОЛОНИИ

Колония муравьев может рассматриваться как многоагентная система, в которой каждый агент (муравей) функционирует автономно по очень простым правилам. В противовес почти примитивному поведению агентов, поведение всей системы получается на удивление разумным [1].

Муравьиные алгоритмы серьезно исследуются учеными с середины 90-х годов. На сегодняшний день уже получены хорошие результаты для оптимизации таких сложных комбинаторных задач, как задача коммивояжера, задача оптимизации маршрутов грузовиков, задача раскраски графа, квадратичная задача о назначениях, задача оптимизации сетевых графиков, задача календарного планирования и многие другие. Особенно эффективны муравьиные алгоритмы при динамической оптимизации процессов в распределенных нестационарных системах, например, трафиков в телекоммуникационных сетях.

2.1. Естественная мотивация

Хотя муравьи слепы, они умеют перемещаться по сложной местности, находить пищу на большом расстоянии от муравейника и успешно возвращаться домой. Выделяя ферменты во время перемещения, муравьи изменяют окружающую среду, обеспечивают коммуникацию, а также отыскивают обратный путь в муравейник.

Самое удивительное в данном процессе – это то, что муравьи умеют находить самый оптимальный путь между муравейником и внешними точками. Чем больше муравьев используют один и тот же путь, тем выше концентрация ферментов на этом пути. Чем ближе внешняя точка к муравейнику, тем больше раз к ней перемещались муравьи. Что касается более удаленной точки, то ее муравьи достигают реже, поэтому по дороге к ней они применяют более сильные ферменты. Чем выше концентрация ферментов на пути, тем предпочтительнее он для муравьев по сравнению с другими доступными. Так муравьиная «логика» позволяет выбирать более короткий путь между конечными точками.

Алгоритмы муравья интересны, поскольку отражают ряд специфических свойств, присущих самим муравьям. Муравьи легко вступают в сотрудничество и работают вместе для достижения общей цели. Алгоритмы муравья работают так же, как муравьи. Это выражается в том, что смоделированные муравьи совместно решают проблему и помогают другим муравьям в дальнейшей оптимизации решения [8].

Рассмотрим пример, представленный на рис. 2.1. Два муравья из муравейника должны добраться до пищи, которая находится за препятствием. Во время перемещения каждый муравей выделяет немного фермента, используя его в качестве маркера [9].

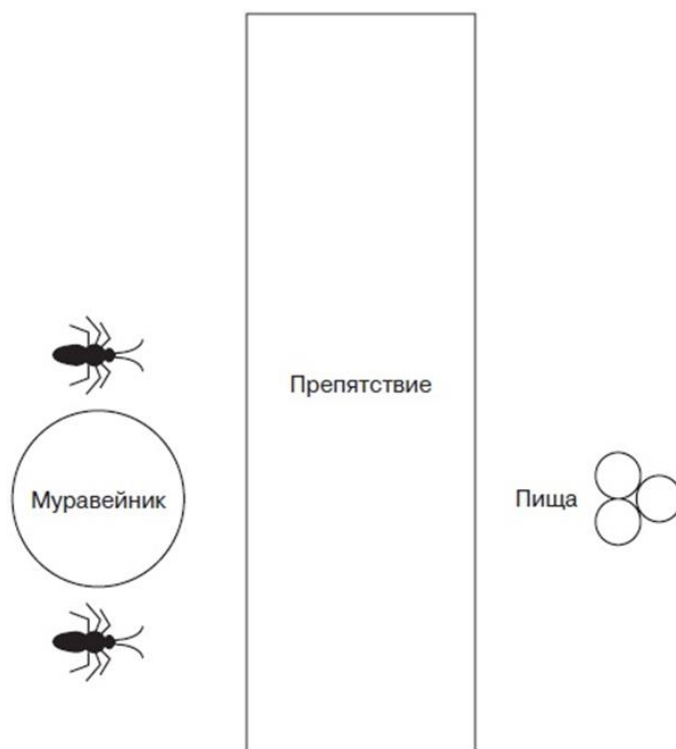


Рис. 2.1. Начальная конфигурация (T_0).

При прочих равных каждый муравей выберет свой путь. Первый муравей выбирает верхний путь, а второй – нижний. Так как нижний путь в два раза короче верхнего, второй муравей достигнет цели за время T_1 . Первый муравей в этот момент пройдет только половину пути (рис. 2.2).

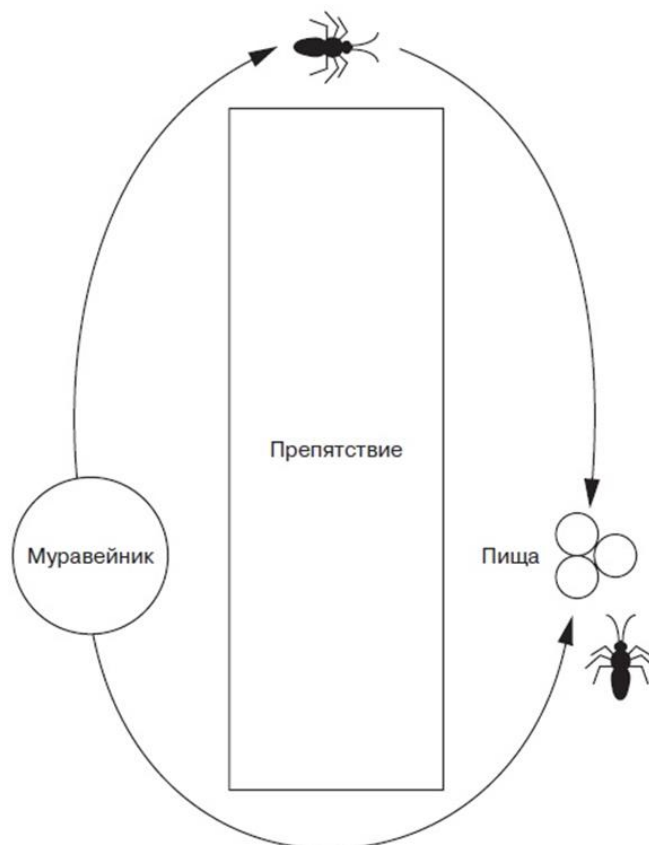


Рис. 2.2. Прошел один период времени (T_1)

Когда один муравей достигает пищи, он берет один из объектов и возвращается к муравейнику по тому же пути. За время T_2 второй муравей вернулся в муравейник с пищей, а первый муравей достиг пищи (рис. 2.3).

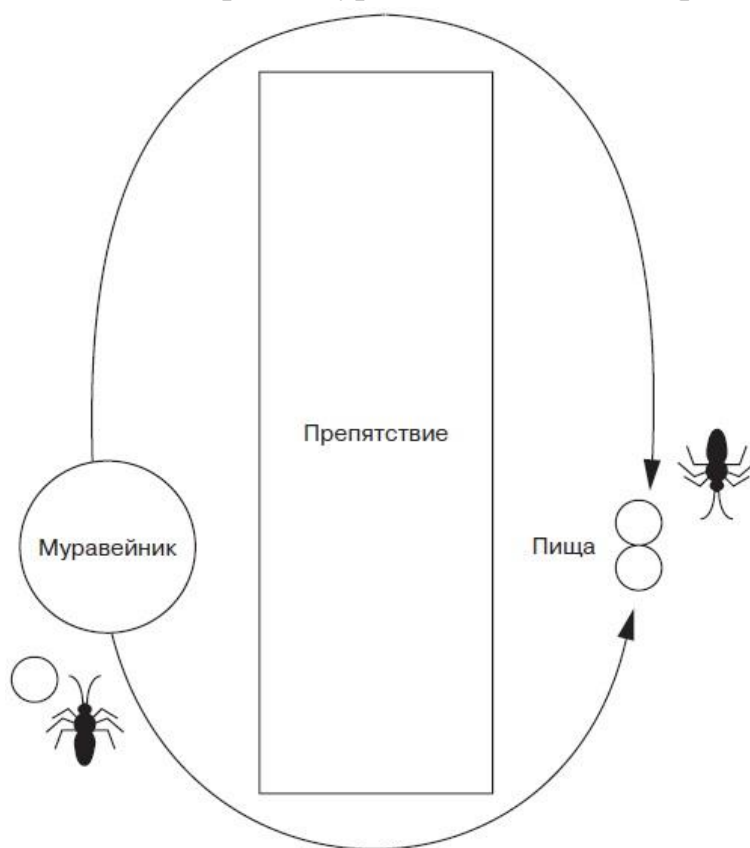


Рис. 2.3. Прошло два периода времени (T_2)

Вспомните, что при перемещении каждого муравья на пути остается немного фермента. Для первого муравья за время $T_0 - T_2$ путь был покрыт ферментом только один раз. В то же самое время второй муравей покрыл путь ферментом дважды. За время T_4 первый муравей вернулся в муравейник, а второй муравей уже успел еще раз сходить к еде и вернуться. При этом концентрация фермента на нижнем пути будет в два раза выше, чем на верхнем. Поэтому первый муравей в следующий раз выберет нижний путь, поскольку там концентрация фермента выше [9].

В этом и состоит базовая идея алгоритма муравья – оптимизация путем не прямой связи между автономными агентами.

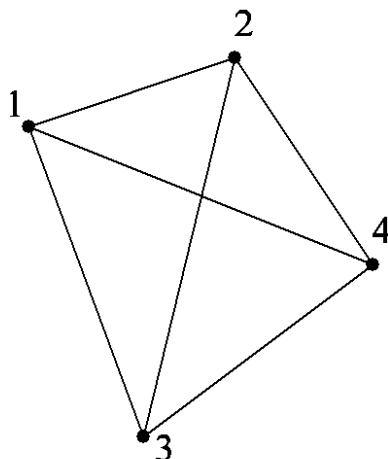
2.2. Описание муравьиного алгоритма

В этом разделе будет подробно рассмотрен алгоритм муравья, чтобы понять, как он работает при решении конкретной проблемы [9].

Граф.

Предположим, что окружающая среда для муравьев представляет собой закрытую двумерную сеть. Сеть – это группа узлов, соединенных посредством граней. Каждая грань имеет вес, который мы обозначим как расстояние между

двумя узлами, соединенными ею. Граф двунаправленный, поэтому муравей может путешествовать по грани в любом направлении (рис. 2.4).



Граф с вершинами $V = \{1, 2, 3, 4\}$
Грани $E = \{\{1, 2\}, \{1, 4\}, \{1, 3\}, \{2, 3\}, \{2, 4\}, \{3, 4\}\}$

Рис. 2.4. Пример полностью замкнутой двумерной сети V с набором граней E
Муравей.

Муравей – это программный агент, который является членом большой колонии и используется для решения какой-либо проблемы. Муравей снабжается набором простых правил, которые позволяют ему выбирать путь в графе. Он поддерживает список табу (tabu list), то есть список узлов, которые он уже посетил. Таким образом, муравей должен проходить через каждый узел только один раз. Путь между двумя узлами графа, по которому муравей посетил каждый узел только один раз, называется путем Гамильтона (Hamiltonian path), по имени математика сэра Уильяма Гамильтона.

Узлы в списке «текущего путешествия» располагаются в том порядке, в котором муравей посещал их. Позже список используется для определения протяженности пути между узлами [10].

Настоящий муравей во время перемещения по пути будет оставлять за собой фермент. В алгоритме муравья агент оставляет фермент на гранях сети после завершения путешествия. О том, как это происходит, рассказывается в разделе «Путешествие муравья».

Начальная популяция

После создания популяция муравьев поровну распределяется по узлам сети. Необходимо равное деление муравьев между узлами, чтобы все узлы имели одинаковые шансы стать отправной точкой. Если все муравьи начнут движение из одной точки, это будет означать, что данная точка является оптимальной для старта, а на самом деле мы этого не знаем.

Движение муравья

Движение муравья основывается на одном и очень простом вероятностном уравнении. Если муравей еще не закончил путь, то есть не посетил все узлы сети, для определения следующей грани пути используется уравнение 2.1:

$$P = \frac{\tau(r,u)^\alpha \times \eta(r,u)^\beta}{\sum_k \tau(r,u)^\alpha \times \eta(r,u)^\beta} \quad (2.1)$$

Здесь $\tau(r,u)$ – интенсивность фермента на грани между узлами r и u , $\eta(r,u)$ – функция, которая представляет измерение обратного расстояния для грани, α – вес фермента, а β – коэффициент эвристики. Параметры α и β определяют относительную значимость двух параметров, а также их влияние на уравнение. Муравей путешествует только по узлам, которые еще не были посещены (как указано списком табу). Поэтому вероятность рассчитывается только для граней, которые ведут к еще не посещенным узлам. Переменная k представляет грани, которые еще не были посещены [10].

Путешествие муравья

Пройденный муравьем путь отображается, когда муравей посетит все узлы диаграммы. Обратите внимание, что циклы запрещены, поскольку в алгоритм включен список табу. После завершения длина пути может быть подсчитана – она равна сумме всех граней, по которым путешествовал муравей. Уравнение 2.2 показывает количество фермента, который был оставлен на каждой грани пути для муравья k . Переменная Q является константой [10].

$$\Delta\tau_{ij}^k = \frac{Q}{L^k(t)} \quad (2.2.)$$

Результат уравнения является средством измерения пути, – короткий путь характеризуется высокой концентрацией фермента, а более длинный путь – более низкой. Затем полученный результат используется в уравнении 2.3, чтобы увеличить количество фермента вдоль каждой грани пройденного муравьем пути (path).

$$\tau_{ij}(t) = \Delta\tau_{ij}(t) + (\tau_{ij}^k(t) \times \rho) \quad (2.3)$$

Данное уравнение применяется ко всему пути, при этом каждая грань помечается ферментом пропорционально длине пути. Поэтому следует дождаться, пока муравей закончит путешествие и только потом обновить уровни фермента, в противном случае истинная длина пути останется неизвестной. Константа ρ – значение между 0 и 1.

Испарение фермента

В начале пути у каждой грани есть шанс быть выбранной. Чтобы постепенно удалить грани, которые входят в худшие пути в сети, ко всем граням

применяется процедура испарения фермента (Pheromone evaporation). Используя константу ρ из уравнения 2.3, мы получаем уравнение 2.4.

$$\tau_{ij}(t) = \tau_{ij}(t) \times (1 - \rho) \quad (2.4)$$

Поэтому для испарения фермента используется обратный коэффициент обновления пути [9].

Повторный запуск

После того как путь муравья завершен, грани обновлены в соответствии с длиной пути и произошло испарение фермента на всех гранях, алгоритм запускается повторно. Список табу очищается, и длина пути обнуляется. Муравьям разрешается перемещаться по сети, основывая выбор грани на уравнении 2.1

Этот процесс может выполняться для постоянного количества путей или до момента, когда на протяжении нескольких запусков не было отмечено повторных изменений. Затем определяется лучший путь, который и является решением.

2.3. Пример расчета итерации

В этом разделе представлено функционирование алгоритма на простом примере, чтобы увидеть, как работают уравнения. Из рис. 2.5 было видно, что это простой сценарий с двумя муравьями, которые выбирают два разных пути для достижения одной цели [9].

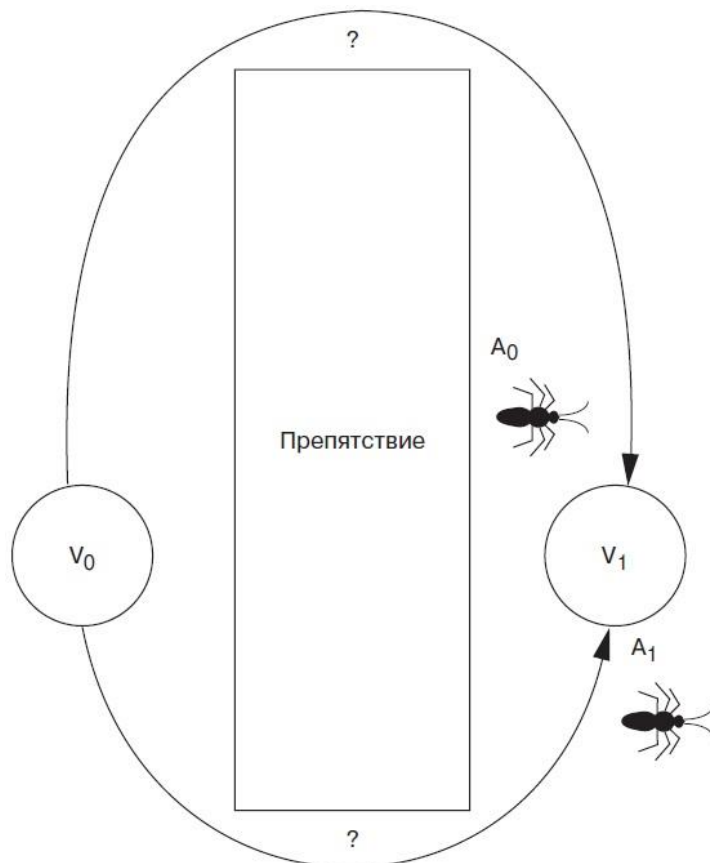


Рис. 2.5. Завершение пути муравья.

На рис. 2.5 показан этот пример с двумя гранями между двумя узлами (V_0 и V_1). Каждая грань инициализируется и имеет одинаковые шансы на то, чтобы быть выбранной.

$$\rho = 0,6$$

$$\alpha = 3,0$$

$$\beta = 1,0$$

	A0	A1
Пройденное расстояние	20	10
Уровень фермента Q	0,5	1,0
Пройденное расстояние		

Для муравья A_1 результат составляет:

$$= 0,1 + (1,0 \times 0,6) = 0,7.$$

Далее с помощью уравнения 2.4 определяется, какая часть фермента испарится и, соответственно, сколько останется. Результат (для каждого пути) составляют:

$$= 0,4 \times (1,0 - 0,6) = 0,16$$

$$= 0,7 \times (1,0 - 0,6) = 0,28.$$

Эти значения представляют новое количество фермента для каждого пути (верхнего и нижнего, соответственно). После перемещения муравьев обратно в узел V_0 воспользуемся вероятностным уравнением выбора пути 1, чтобы определить, какой путь должны выбрать муравьи.

Вероятность того, что муравей выберет верхний путь (представленный количеством фермента 0,16), составляет:

$$\frac{(0,16)^{3,0} \times (0,5)^{1,0}}{((0,16)^{3,0} \times (0,5)^{1,0}) + ((0,28)^{3,0} \times (1,0)^{1,0})} = \frac{0,002048}{0,024} = P(0,085)$$

Вероятность того, что муравей выберет нижний путь (представленный количеством фермента 0,28), составляет:

$$\frac{(0,28)^{3,0} \times (1,0)^{1,0}}{((0,16)^{3,0} \times (0,5)^{1,0}) + ((0,28)^{3,0} \times (1,0)^{1,0})} = \frac{0,021952}{0,024} = P(0,915)$$

При сопоставлении двух вероятностей оба муравья выберут нижний путь, который является наиболее оптимальным.

2.4. Программная реализация

В качестве примера будет рассмотрена задача коммивояжера (Traveling Salesman Problem – TSP). Она заключается в том, чтобы найти кратчайший путь между городами, при котором каждый город будет посещен всего один раз. То есть надо найти кратчайший Гамильтонов путь в графе, где в качестве узлов выступают города, а в качестве граней – соединяющие их дороги. Математики впервые изучили задачу TSP в 1930е гг., в частности, ею занимался Карл Менгер (Karl Menger) в Вене. Следует отметить, что похожие задачи ис-

следовались еще в 19 в. ирландским математиком сэром Уильямом Роуэном Гамильтоном (Sir William Rowan Hamilton) [9].

Исходный код.

В приведенных листингах, написанных на языке программирования C++, иллюстрируется алгоритм муравья, который используется для поиска оптимальных решений задачи коммивояжера.

Для начала представлена структура данных как для городов, так и для агентов (муравьев), которые будут по ним путешествовать. Листинг 2.1 включает типы данных и символьные константы, которые используются для представления городов и муравьев [9].

```
#define MAX_CITIES    30
#define MAX_DISTANCE    100

#define MAX_TOUR    (MAX_CITIES * MAX_DISTANCE)

typedef struct {
    int x;
    int y;
} cityType;
#define MAX_ANTS    30

typedef struct {
    int curCity;
    int nextCity;
    unsigned char tabu[MAX_CITIES];
    int pathIndex;
    unsigned char path[MAX_CITIES];
    double tourLength;
} antType;
```

Листинг 2.1. Типы данных, символьные константы для представления городов и муравьев.

Структура `cityType` используется для определения города в матрице `MAX_DI. STANCE` на `MAX_DISTANCE`. Структура `antType` представляет одного муравья. Кроме отслеживания текущего и следующего города на пути (поля `curCity` и `next City`) каждый муравей также должен учитывать города, кото-

рые уже были посещены (массив `tabu`) и длину пройденного пути. Наконец, общая длина пути сохраняется в поле `tourLength`.

В зависимости от размера задачи TSP иногда полезно изменить параметры в уравнениях. Ниже приводятся значения параметров по умолчанию для задачи TSP с 30 городами (листинг 2.2). Изменения параметров будут описаны в следующих разделах.

```
#define ALPHA    1.0
#define BETA      5.0
#define RHO      0.5    /* Интенсивность / Испарение */
#define QVAL     100

#define MAX_TOURS 20

#define MAX_TIME  (MAX_TOURS * MAX_CITIES)

#define INIT_PHEROMONE (1.0 / MAX_CITIES)
```

Листинг 2.2. Параметры задачи, для решения которой используется алгоритм муравья.

Параметр ALPHA (α) определяет относительную значимость пути (количество фермента на пути). Параметр BETA (β) устанавливает относительную значимость видимости (обратной расстоянию). Параметр RHO (ρ) используется как коэффициент количества фермента, которое муравей оставляет на пути (этот параметр также известен как продолжительность пути), где $(1,0 - \rho)$ показывает коэффициент испарения на пути после его завершения. Параметр QVAL (или просто Q в уравнении 2.2) – это константа, относящаяся к количеству фермента, которое было оставлено на пути. Остальные константы будут рассмотрены при описании исходного кода алгоритма.

Структуры данных в алгоритме включают массивы `cities` и `ants`. Другой специальный двумерный массив, `distance`, содержит предварительно рассчитанные расстояния от одного города до всех других. Уровни фермента сохраняются в массиве `pheromone`. Каждый двумерный массив в алгоритме использует первый индекс в качестве начала грани, то есть исходного города, а второй – в качестве конечного. Все глобальные структуры данных представлены в листинге 2.3, а функция инициализации – в листинге 2.4 [9].

```
cityType cities[MAX_CITIES];
```



```

antType ants[MAX_ANTS];

        /* Из      В      */
double distance[MAX_CITIES][MAX_CITIES];

        /* Из      В      */
double pheromone[MAX_CITIES][MAX_CITIES];

double      best=(double)MAX_TOUR;
int          bestIndex;

```

Листинг 2.3. Глобальные структуры данных.

Сначала будет рассмотрена функция init (листинг 2.4).

```

void init( void )
{
    int from, to, ant;

    /* Создание городов */
    for (from = 0 ; from < MAX_CITIES ; from++) {

        /* Случайным образом распределяем города */
        cities[from].x = getRand( MAX_DISTANCE );
        cities[from].y = getRand( MAX_DISTANCE );

        for (to = 0 ; to < MAX_CITIES ; to++) {
            distance[from][to] = 0.0;
            pheromone[from][to] = INIT_PHEROMONE;
        }
    }

    /* Вычисляем расстояние между городами */
    for ( from = 0 ; from < MAX_CITIES ; from++) {
        for ( to = 0 ; to < MAX_CITIES ; to++) {
            if ((to != from) && (distance[from][to] == 0.0)) {
                int xd = abs(cities[from].x - cities[to].x);
                int yd = abs(cities[from].y - cities[to].y);

```

```

        distance[from][to] = sqrt( (xd * xd) + (yd * yd) );
        distance[to][from] = distance[from][to];
    }
}

/* Инициализация муравьев */
to = 0;
for ( ant = 0 ; ant < MAX_ANTS ; ant++ ) {

    /* Распределяем муравьев по городам равномерно */
    if (to == MAX_CITIES) to = 0;
    ants[ant].curCity = to++;
    for ( from = 0 ; from < MAX_CITIES ; from++ ) {
        ants[ant].tabu[from] = 0;
        ants[ant].path[from] = -1;
    }

    ants[ant].pathIndex = 1;
    ants[ant].path[0] = ants[ant].curCity;
    ants[ant].nextCity = -1;
    ants[ant].tourLength = 0.0;

    /* Помещаем город, в котором находится муравей, в список табу */
    ants[ant].tabu[ants[ant].curCity] = 1;
}
}

```

Листинг 2.4. Функция инициализации.

Функция `init` выполняет три базовых действия, которые требуются для подготовки алгоритма муравья. Первое действие – это создание городов. Для каждого города, который требуется создать (количество задано константой `MAX_CITIES`) генерируются произвольные числа для координат по осям `x` и `y`, которые затем сохраняются в запись текущего города. Кроме того, во время создания городов одновременно очищаются массивы `distance` и `pheromone`.

Далее, немного оптимизируя алгоритм, мы выполняем предварительный расчет расстояний между всеми городами, которые были созданы. Расстояние

вычисляется с помощью обычной двумерной геометрии координат (теорема Пифагора) [9].

Наконец, инициализируется массив `ant`. Вспомните, что муравьи должны быть равномерно распределены по всем городам. Для этого используется переменная `to` в качестве счетчика псевдоцикла. Когда значение переменной `to` становится равным номеру последнего города, она возвращается к первому городу (городу 0), и процесс повторяется. Поле `curCity` в структуре `ant` устанавливается в значение текущего города (первого города для муравья). Затем очищаются списки `tabu` и `path`. Ноль в массиве `tabu` обозначает, что город еще не был посещен; единица свидетельствует, что город уже включен в путь муравья. В массив `path` помещается номер текущего города для муравья, и очищается поле `tourLength` (длина пути).

После завершения пути каждый муравей должен быть повторно инициализирован, а затем распределен по графу. Это выполняет функция `restartAnts` (листинг 2.5) [9].

```
void restartAnts( void )
{
    int ant, i, to=0;

    for ( ant = 0 ; ant < MAX_ANTS ; ant++ ) {

        if ( ants[ant].tourLength < best ) {
            best = ants[ant].tourLength;
            bestIndex = ant;
        }

        ants[ant].nextCity = -1;
        ants[ant].tourLength = 0.0;
        for ( i = 0 ; i < MAX_CITIES ; i++ ) {
            ants[ant].tabu[i] = 0;
            ants[ant].path[i] = -1;
        }
        if ( to == MAX_CITIES ) to = 0;
        ants[ant].curCity = to++;
        ants[ant].pathIndex = 1;
        ants[ant].path[0] = ants[ant].curCity;
```

```

ants[ant].tabu[ants[ant].curCity] = 1;

}

}

```

Листинг 2.5. Функция restartAnts предназначена для повторной инициализации всех муравьев.

Во время повторной инициализации самая оптимальная длина пути сохраняется, чтобы можно было оценить прогресс муравьев. Затем структура алгоритма очищается и повторно инициализируется – начинается следующее путешествие.

Функция selectNextCity позволяет выбрать следующий город для составления пути. Она вызывает функцию antProduct, которая используется для расчета уравнения 2.1 (листинг 2.6). Функция pow (возведение числа x в степень y) является частью стандартной математической библиотеки [9].

```

double antProduct( int from, int to )
{
    return (( pow( pheromone[from][to], ALPHA ) *
                pow( (1.0 / distance[from][to]), BETA ) ));
}

```

```

int selectNextCity( int ant )
{
    int from, to;
    double denom=0.0;

    /* Выбрать следующий город */
    from = ants[ant].curCity;

    /* Расчет знаменателя */
    for (to = 0 ; to < MAX_CITIES ; to++) {
        if(ants[ant].tabu[to] == 0) {
            denom += antProduct( from, to );
        }
    }
}

```

```

assert(denom != 0.0);

```

```

do {

    double p;

    to++;
    if (to >= MAX_CITIES) to = 0;

    if ( ants[ant].tabu[to] == 0 ) {
        p = antProduct(from, to)/denom;
        if (getSRand() < p ) break;
    }

} while (1);
return to;
}

```

Листинг 2.6. Функция antProduct и selectNextCity.

Функция selectNextCity вызывается для заданного муравья и определяет, которую из граней выбрать в соответствии со списком tabu. Выбор грани основывается на вероятностном уравнении 2.1, которое вычисляет коэффициент за данного пути от суммы других путей. Первой задачей функции selectNextCity является расчет знаменателя выражения. После этого все грани, которые еще не были выбраны, проверяются с помощью уравнения 2.1, чтобы муравей мог выбрать путь. Как только грань найдена, функция определяет город, к которому перейдет муравей.

Следующая функция, simulateAnts, рассчитывает движения муравьев по графу (листинг 2.7) [9].

```

int simulateAnts( void )
{
    int k;
    int moving = 0;

    for (k = 0 ; k < MAX_ANTS ; k++) {

        /* Убедиться, что муравью есть куда идти */
        if (ants[k].pathIndex < MAX_CITIES) {

```

```

    ants[k].nextCity = selectNextCity( k );
    ants[k].tabu[ants[k].nextCity] = 1;
    ants[k].path[ants[k].pathIndex++] = ants[k].nextCity;

    ants[k].tourLength += distance[ants[k].curCity][ants[k].nextCity];

    /* Обработка окончания путешествия (из последнего города в первый
*/
    if (ants[k].pathIndex == MAX_CITIES) {
        ants[k].tourLength +=
            distance[ants[k].path[MAX_CITIES-1]][ants[k].path[0]];
    }

    ants[k].curCity = ants[k].nextCity;

    moving++;
}

return moving;
}

```

Листинг 2.7. Функция simulateAnts.

Функция simulateAnts проходит по массиву ant и перемещает муравьев из текущего города в новый город, предлагаемый вероятностным уравнением 2.1. Программа отмечает поле pathIndex, чтобы убедиться, что муравей еще не закончил путь. После этого вызывается функция selectNextCity, которая определяет следующую грань. Затем выбранный город загружается в структуру ant в поле nextCity, а также в списки path и tabu. Рассчитывается значение tourLength, чтобы сохранить длину пути. Наконец, если достигнут конец пути, в значение tourLength добавляется расстояние до начального города. Это позволяет построить полный путь через все остальные города и вернуться в начальный город.

Один вызов функции simulateAnts позволяет каждому муравью перейти от одного города к другому. Функция simulateAnts возвращает нуль, если путь уже завершен, а в противном случае – значение, отличное от нуля.

После завершения пути выполняется процесс обновления путей. Он включает не только обновление путей по количеству фермента, оставленного муравьями, но и существующего фермента, часть которого испарилась. Функция `updateTrails` выполняет эту часть алгоритма (листинг 2.8) [9].

```
void updateTrails( void )
{
    int from, to, i, ant;

    /* Испарение фермента */
    for (from = 0 ; from < MAX_CITIES ; from++) {

        for (to = 0 ; to < MAX_CITIES ; to++) {

            if (from != to) {

                pheromone[from][to] *= (1.0 - RHO);

                if (pheromone[from][to] < 0.0) pheromone[from][to] = INIT_PHEROMONE;

            }

        }

    }

    /* Нанесение нового фермента */

    /* Для пути каждого муравья */
    for (ant = 0 ; ant < MAX_ANTS ; ant++) {

        /* Обновляем каждый шаг пути */
        for (i = 0 ; i < MAX_CITIES ; i++) {

            if (i < MAX_CITIES-1) {
                from = ants[ant].path[i];
                to = ants[ant].path[i+1];
```

```

        } else {
            from = ants[ant].path[i];
            to = ants[ant].path[0];
        }

        pheromone[from][to] += (QVAL / ants[ant].tourLength);
        pheromone[to][from] = pheromone[from][to];

    }
}
for (from = 0 ; from < MAX_CITIES ; from++) {
    for (to = 0 ; to < MAX_CITIES ; to++) {
        pheromone[from][to] *= RHO;
    }
}
}

```

Листинг 2.8. Функция updateTrails – испарение и размещение нового фермента.

Первая задача функции updateTrails заключается в том, чтобы «испарить» часть фермента, который уже находится на пути. Она выполняется на всех границах с помощью уравнения 2.4. Следующая задача – получение нового фермента на путях, пройденных муравьями во время последних перемещений. Для этого необходимо пройти по элементам массива ant и, руководствуясь значением поля path, «распылить» новый фермент на посещенной муравьем грани в соответствии с уравнением 2.2. Затем следует применить параметр RHO, чтобы снизить интенсивность фермента, который выделяют муравьи [9].

```

void emitDataFile( int ant )
{
    int city;
    FILE *fp;

    fp = fopen("cities.dat", "w");
    for (city = 0 ; city < MAX_CITIES ; city++) {
        fprintf(fp, "%d %d\n", cities[city].x, cities[city].y);
    }
    fclose(fp);
}

```



```

    fp = fopen("solution.dat", "w");
    for (city = 0 ; city < MAX_CITIES ; city++) {
        fprintf(fp, "%d %d\n",
            cities[ ants[ant].path[city] ].x,
            cities[ ants[ant].path[city] ].y );
    }
    fprintf(fp, "%d %d\n",
        cities[ ants[ant].path[0] ].x,
        cities[ ants[ant].path[0] ].y );
fclose(fp);
}

void emitTable( void )
{
    int from, to;

    for (from = 0 ; from < MAX_CITIES ; from++) {
        for (to = 0 ; to < MAX_CITIES ; to++) {
            printf("%5.2g ", pheromone[from][to]);
        }
        printf("\n");
    }
    printf("\n");
}

```

Листинг 2.9. Функция emitDataFile.

Функция emitDataFile в листинге 2.9 предназначена для создания наглядных графиков перемещения муравьев по городам.

Следующая функция main очень проста (листинг 2.10). После инициализации генератора случайных чисел и установки начальных значений параметров алгоритма запускается симуляция для количества шагов, заданного константой MAX_TIME. Когда муравей завершает путь (который он проходит с помощью функции simulateAnts), вызывается функция updateTrails для того, чтобы изменить окружающую среду [9].

```

int main()
{

```

```

int curTime = 0;
srand( time(NULL) );
init();

while (curTime++ < MAX_TIME) {
    if ( simulateAnts() == 0 ) {
        updateTrails();

        if (curTime != MAX_TIME)
            restartAnts();

        printf("Время %d (%g)\n", curTime, best);
    }
}

printf("Лучший путь %g\n", best);
printf("\n\n");

return 0;
}

```

Листинг 2.10. Функция main для симуляции алгоритма муравья.

Обратите внимание, что каждый раз при завершении пути вызывается функция restartAnts. Это делается для того, чтобы подготовиться к следующему путешествию по диаграмме. Функция restartAnts не вызывается только в том случае, если симуляция уже завершена. Дело в том, что функция уничтожает информацию о пути муравья, которую необходимо сохранить в самом конце, чтобы определить наилучший путь.

Пример запуска.

Теперь рассмотрим несколько примеров запуска алгоритма муравья для задачи коммивояжера.

При первом запуске выполняется решение задачи для 30 городов (рис. 2.6). Были заданы следующие параметры: $\alpha = 1,0$, $\beta = 5,0$, $\rho = 0,5$, $Q = 100$.

При втором запуске выполняется решение задачи для 50 городов (рис. 2.7). Для нее были заданы те же параметры, что и для предыдущей.

Решение задачи в первом и втором случае было найдено быстрее, чем за пять путешествий муравьев. При каждом запуске количество муравьев равнялось количеству городов.

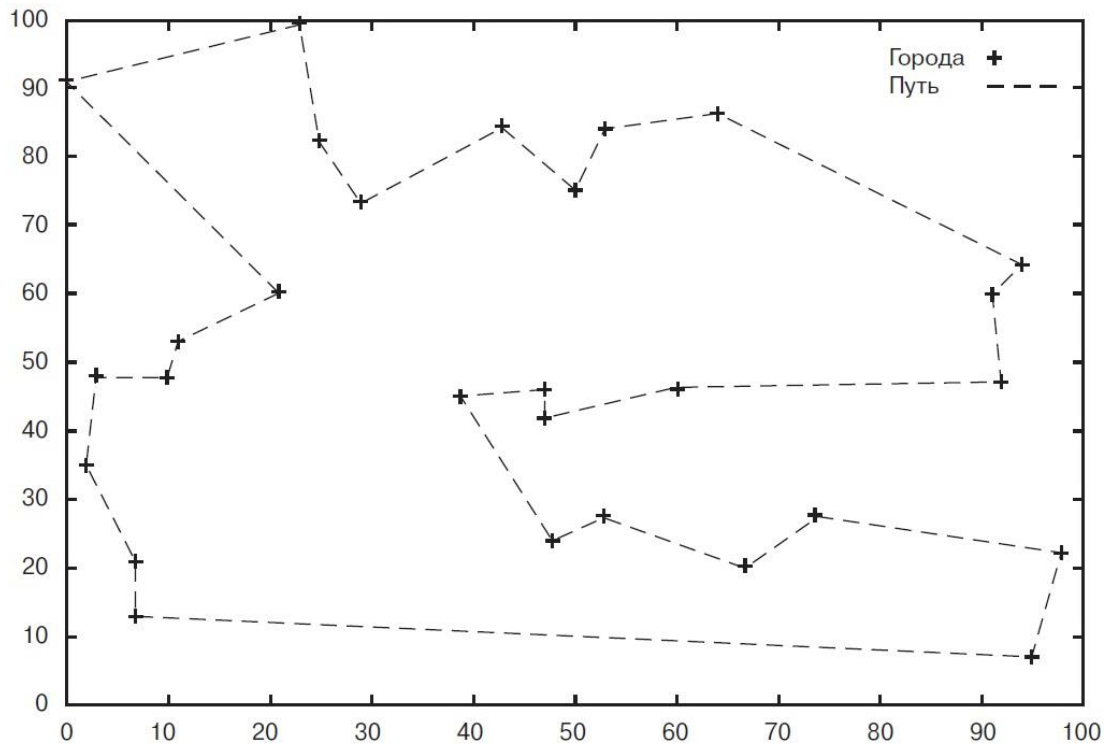


Рис. 2.6. Пример решения проблем TSP для 30 городов.

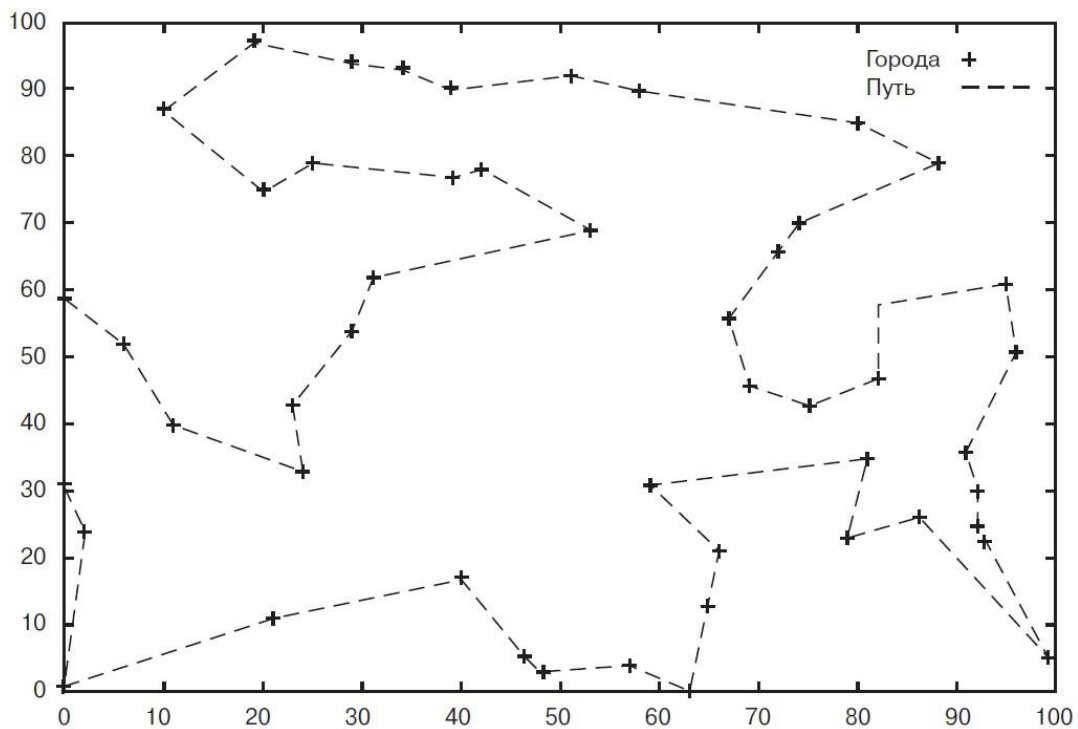


Рис. 2.7. Пример решения проблемы TSP для 50 городов.

На рисунке 2.8 представлен результат компиляции программы алгоритма муравья. Результат показывает наиболее оптимальное решение задачи коммивояжера.



 stdout	 copy
Время 30 (542.121)	
Время 60 (496.318)	
Время 90 (493.537)	
Время 120 (493.45)	
Время 150 (478.09)	
Время 180 (478.09)	
Время 210 (478.09)	
Время 240 (472.977)	
Время 270 (472.977)	
Время 300 (472.977)	
Время 330 (472.977)	
Время 360 (472.977)	
Время 390 (472.977)	
Время 420 (472.977)	
Время 450 (472.977)	
Время 480 (472.977)	
Время 510 (472.977)	
Время 540 (472.977)	
Время 570 (472.977)	
Время 600 (472.977)	
Лучший путь 472.977	

Рис. 2.8. Результат выполнения программы.

2.5. Обзор модификаций муравьиного алгоритма

Результаты первых экспериментов с применением муравьиного алгоритма для решения задачи коммивояжера были многообещающими, однако далеко не лучшими по сравнению с уже существовавшими методами. Однако простота классического муравьиного алгоритма (названного «муравьиной системой») оставляла возможности для доработок – и именно алгоритмические усовершенствования стали предметом дальнейших исследований в области комбинаторной оптимизации. В основном, эти усовершенствования связаны с большим использованием истории поиска и более тщательным исследованием областей вокруг уже найденных удачных решений. Ниже рассмотрены наиболее примечательные из модификаций [10].

Elitist Ant System

Одним из таких усовершенствований является введение в алгоритм так называемых «элитных муравьев». Опыт показывает, что, проходя ребра, входящие в короткие пути, муравьи с большей вероятностью будут находить еще более короткие пути. Таким образом, эффективной стратегией является искусственное увеличение уровня феромонов на самых удачных маршрутах. Для это-

го на каждой итерации алгоритма каждый из элитных муравьев проходит путь, являющийся самым коротким из найденных на данный момент.

Эксперименты показывают, что, до определенного уровня, увеличение числа элитных муравьев является достаточно эффективным, позволяя значительно сократить число итераций алгоритма. Однако, если число элитных муравьев слишком велико, то алгоритм достаточно быстро находит субоптимальное решение и застревает в нем. Как и другие изменяемые параметры, оптимальное число элитных муравьев следует определять опытным путем.

Ant-Q

Лука Гамбарделла и Марко Дориго опубликовали в 1995 году работу, в которой они представили муравьиный алгоритм, получивший свое название по аналогии с методом машинного обучения Q-learning. В основе алгоритма лежит идея о том, что муравьиную систему можно интерпретировать как систему обучения с подкреплением. Ant-Q усиливает эту аналогию, заимствуя многие идеи из Q-обучения.

Алгоритм хранит Q-таблицу, сопоставляющую каждому из ребер величину, определяющую «полезность» перехода по этому ребру. Эта таблица изменяется в процессе работы алгоритма – то есть обучения системы. Значение полезности перехода по ребру вычисляется исходя из значений полезностей перехода по следующим ребрам в результате предварительного определения возможных следующих состояний. После каждой итерации полезности обновляются исходя из длин путей, в состав которых были включены соответствующие ребра.

Ant Colony System

В 1997 году те же исследователи опубликовали работу, посвященную еще одному разработанному ими муравьиному алгоритму. Для повышения эффективности по сравнению с классическим алгоритмом, ими были введены три основных изменения.

Во-первых, уровень феромонов на ребрах обновляется не только в конце очередной итерации, но и при каждом переходе муравьев из узла в узел. Во-вторых, в конце итерации уровень феромонов повышается только на кратчайшем из найденных путей. В-третьих, алгоритм использует измененное правило перехода: либо, с определенной долей вероятности, муравей безусловно выбирает лучшее – в соответствии с длиной и уровнем феромонов – ребро, либо производит выбор так же, как и в классическом алгоритме.

Max-min Ant System

В том же году Томас Штютцле и Хольгер Хоос предложили муравьиный алгоритм, в котором повышение концентрации феромонов происходит только на лучших путях из пройденных муравьями. Такое большое внимание к локальным оптимумам компенсируется вводом ограничений на максимальную и минимальную концентрацию феромонов на ребрах, которые крайне эффективно защищают алгоритм от преждевременной сходимости к субоптимальным решениям.

На этапе инициализации, концентрация феромонов на всех ребрах устанавливается равной максимальной. После каждой итерации алгоритма только один муравей оставляет за собой след – либо наиболее успешный на данной итерации, либо, аналогично алгоритму с элитизмом, элитный. Этим достигается, с одной стороны, более тщательное исследование области поиска, с другой – его ускорение.

ASrank

Бернд Бульнхаймер, Рихард Хартл и Кристине Штраусс разработали модификацию классического муравьиного алгоритма, в котором в конце каждой итерации муравьи ранжируются в соответствии с длинами пройденных ими путей. Количество феромонов, оставляемого муравьем на ребрах, таким образом, назначается пропорционально его позиции. Кроме того, для более тщательного исследования окрестностей уже найденных удачных решений, алгоритм использует элитных муравьев.

Алгоритм муравья может применяться для решения многих задач, таких как распределение ресурсов и работы. При решении задачи распределения ресурсов (Quadratic Assignment Problem – QAP) необходимо задать группу ресурсов n для ряда адресатов m и при этом минимизировать расходы на перераспределение (то есть функция должна найти наилучший способ распределения ресурсов). Обнаружено, что алгоритм муравья дает решения такого же качества, как и другие, более стандартные способы.

Намного сложнее проблема распределения работы (Jobshop Sheduling Problem – JSP). В этой задаче группа машин M и заданий J (состоящих из последовательности действий, осуществляемых на машинах) должны быть распределены таким образом, чтобы все задания выполнялись за минимальное время. Хотя решения, найденные с помощью алгоритма муравья, не являются оптимальными, применение алгоритма для данной проблемы показывает, что с его помощью можно решать аналогичные задачи. Алгоритм муравья применяется для решения других задач, например, прокладки маршрутов для автомобилей, расчета цветов для графиков и маршрутизации в сетях.

3. АЛГОРИТМ ПЧЕЛИНОЙ КОЛОНИИ

Алгоритм пчелиной колонии – один из полиномиальных эвристических алгоритмов для решения оптимизационных задач в области информатики и исследования операций. Относится к категории стохастических бионических алгоритмов, основан на имитации поведения колонии медоносных пчел при сборе нектара в природе [11].

3.1. Естественная мотивация

В обычной колонии пчел, например, *Apis mellifera* (медоносная пчела домашняя), предполагается, что пчелы со временем выполняют разные роли. В типичном улье может быть от 5000 до 20 000 особей. Взрослые особи (в возрасте от 20 до 40 дней), как правило, становятся фуражирами (foragers). Фуражиры обычно выполняют одну из трех ролей: активные фуражиры, фуражиры-разведчики и неактивные фуражиры [12].

Активные фуражиры летят к источнику нектара, обследуют соседние источники, собирают нектар и возвращаются в улей. Разведчики обследуют местность вокруг улья (площадью до 50 квадратных миль) в поисках новых источников нектара. Примерно 10% пчел-фуражиров в улье задействованы в качестве разведчиков (рис. 3.1.).

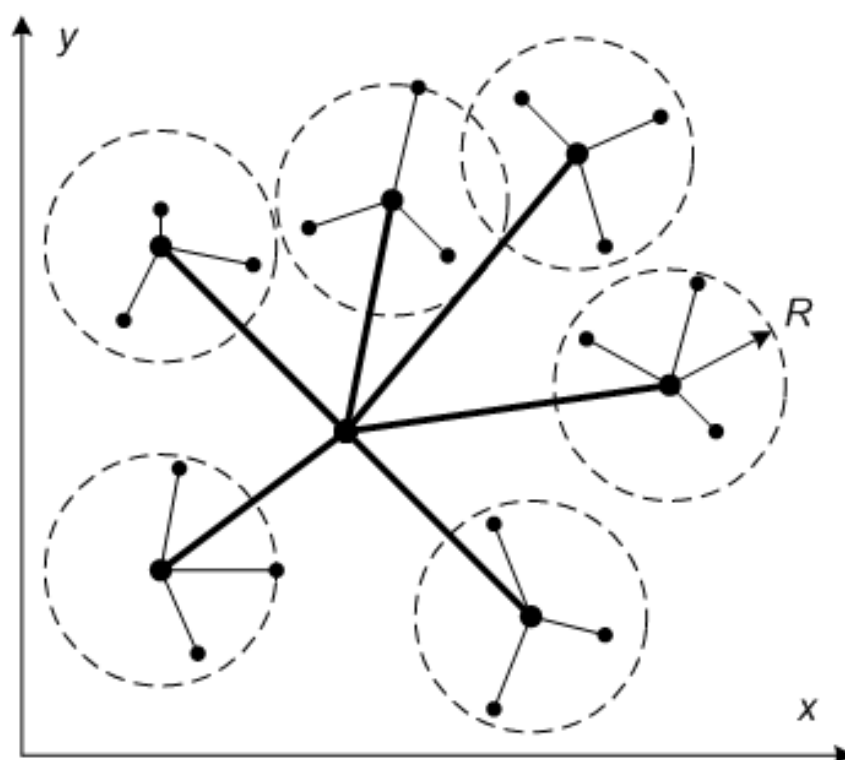


Рис. 3.1. Схематичное изображение стратегии разведки двумерного пространства (жирные линии — вылеты разведчиков, тонкие линии — уточнение решений рабочими пчелами)

В любой момент некоторое количество пчел-фуражиров неактивно. Они ждут неподалеку от входа в улей. Когда активные фуражиры и разведчики возвращаются в улей, то – в зависимости от качества источника нектара, который они только что посетили, - они могут исполнять виляющий танец (waggle dance) перед ждущими неактивными пчелами. Есть довольно веские доказательства того, что этот виляющий танец несет информацию неактивным пчелам о местонахождении и качестве источника нектара. Неактивные фуражиры извлекают из виляющего танца эту информацию об источниках нектара и могут становиться активными фуражирами [12].

В целом, активный фуражир продолжает собирать нектар из конкретного источника до тех пор, пока он не истощится, после чего эта пчела становится неактивным фуражиром.

Исходя, из природного посыла, можно представить, что расположение глобального экстремума – это участок, где больше всего нектара, причем этот участок единственный, то есть в других местах нектар есть, но меньше. А пчелы живут не на плоскости, где для определения месторасположения участков достаточно знать две координаты, а в многомерном пространстве, где каждая координата представляет собой один параметр функции, которую надо оптимизировать. Найденное количество нектара представляет собой значение целевой функции в этой точке (в случае, если мы ищем глобальный максимум, если мы ищем глобальный минимум, то целевую функцию достаточно умножить на -1). Таким образом, пчелы могут собирать и отрицательное количество нектара.

3.2. Описание пчелиного алгоритма

Рассмотрим более подробно алгоритм применительно к нахождению экстремумов функции. При этом будем считать, что отыскивается глобальный максимум функции. В алгоритме каждое решение представляется в виде пчелы, которая знает (хранит) расположение (координаты или параметры многомерной функции) какого-то участка поля, где можно добыть нектар.

В начале алгоритма в точки, описываемые случайными координатами, отправляется некоторое количество пчел-разведчиков (пусть будет S пчел, от слова scout). Таким образом [13]:

1 шаг: Необходимо задать количество пчел-разведчиков S . В точки со случайными координатами $X_{\beta,0} \in D$, отправляются пчелы-разведчики, где β – номер пчелы разведчика, $\beta \in [1: S]$, а 0 обозначает номер итерации в данный момент времени. Считаются значения целевой функции $F(X)$ в этих точках.

2 шаг: В области D с помощью полученных значений выделяют два вида участков (подобластей) d_β .

Первый вид содержит n лучших участков, которые соответствуют наибольшим или наименьшим значениям целевой функции, в зависимости от того решается задача на минимум или на максимум функции.

Второй m перспективных участков, соответствующих значениям целевой функции, наиболее близким к наилучшим значениям.

Подобласть d_β является подобластью локального поиска, представляющая собой гиперкуб в пространстве R^k с центром в точке $X_{\beta,0}$. Длина его сторон равна 2Δ , где Δ – параметр, называемый размером области локального поиска.

3 шаг: Сравнивается евклидово расстояние $\|X_{\beta,0} - X_{\gamma,0}\|$ между двумя агентами-разведчиками. Для точек $A = (x_1, x_2, \dots, x_n)$ и $B = (y_1, y_2, \dots, y_n)$ евклидово расстояние считается по формуле

$$d(A, B) = \sqrt{(x_1 - y_1)^2 + (x_2 - y_2)^2 + \dots + (x_n - y_n)^2}$$

Если евклидово расстояние оказывается меньше фиксированной величины, то возможны два следующих варианта метода [14]:

- поставить в соответствие этим агентам два различных пересекающихся участка d_β, d_γ (лучших и/или перспективных);
- поставить в соответствие тем же агентам один участок, центр которого находится в точке, соответствующей агенту с большим значением целевой функции. Из этих двух вариантов в работе используется второй вариант.

4 шаг: В каждый из лучших и перспективных участков посылаются по N и по M агентов, соответственно. Координаты этих агентов в указанных участках определяются случайным образом.

5 шаг: В полученных точках снова считается значение целевой функции $F(X)$, снова выбирается наибольшее или наименьшее значение. Точка, в которой значение функции является максимальным, становится центром новой подобласти.

6 шаг: Шаги 4 и 5 повторяются до тех пор, пока не будет получен искомый результат, если такой известен, либо до тех пор, пока полученные значения координат экстремумов и значений функции в них не повторятся τ раз, где τ — параметр останова.

Так же данный алгоритм, для нахождения максимума функции можно представить в виде блок-схемы представленной на рис. 3.2 (а, б).

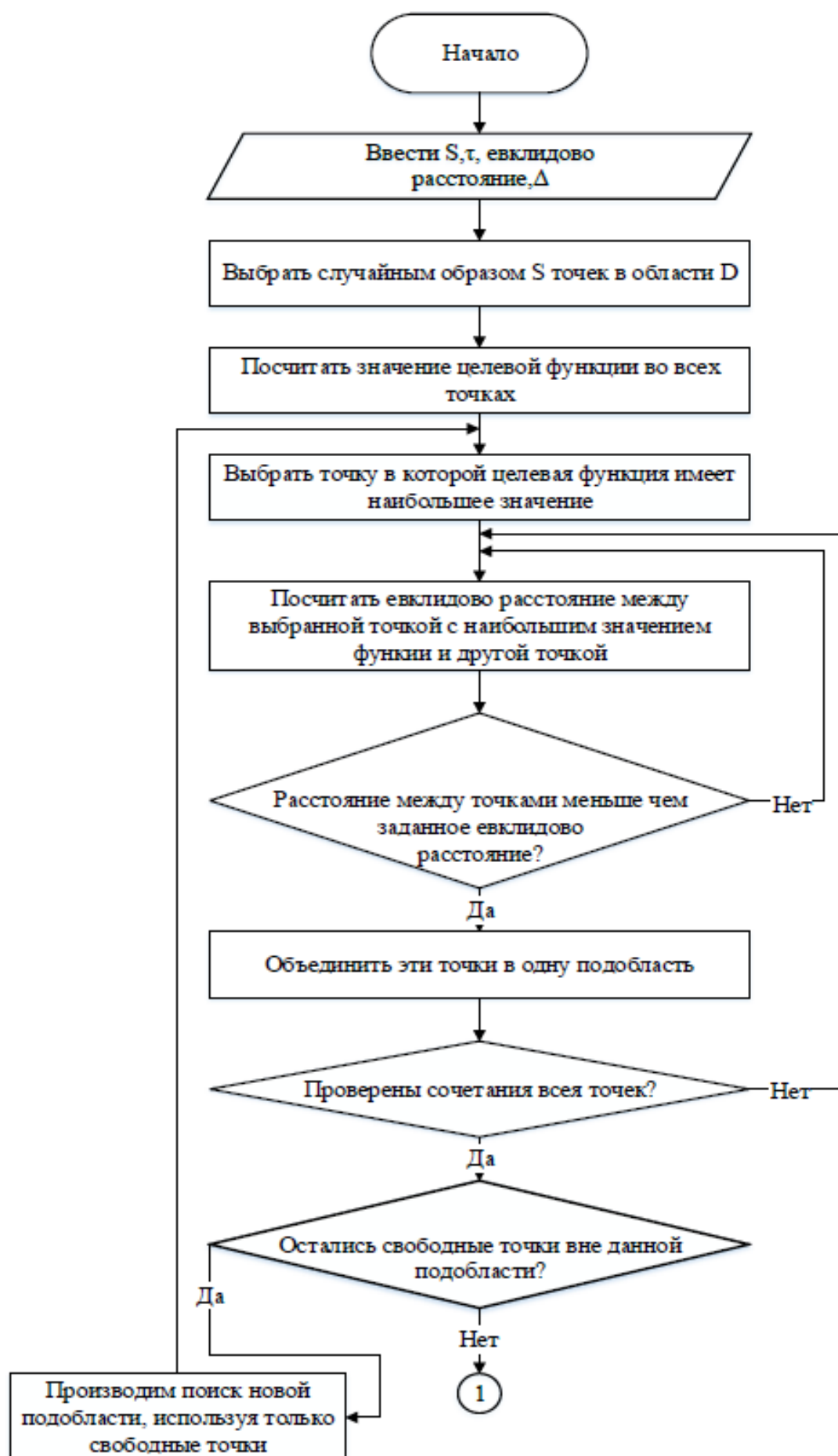


Рис. 3.2. (а). Блок-схема метода пчелиного роя

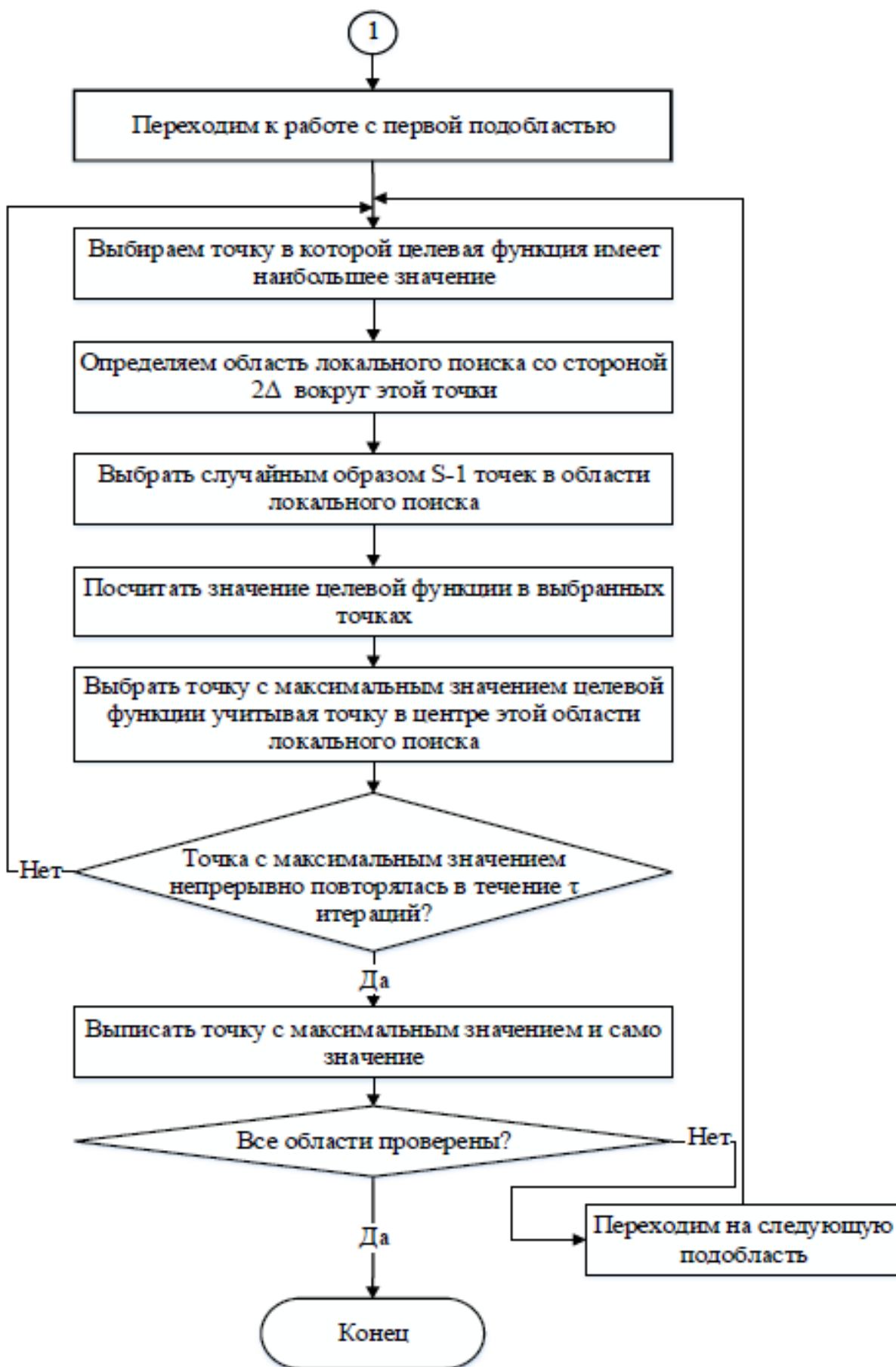


Рис. 3.2. (б). Продолжение блок-схемы метода пчелиного роя

3.3. Пример расчета итерации

Пусть в качестве целевой функции у нас выступает функция

$$f(x, y) = -(x^2 + y^2)$$

Исходник

Знак «-» в данном случае стоит, чтобы у функции был глобальный максимум, а не минимум. Известно, что глобальный (и единственный) максимум этой функции находится в точке $(0; 0)$, причем $f(0, 0) = 0$.

Зафиксируем необходимые параметры [15]:

- Количество пчел-разведчиков: 10;
- Количество пчел, отправляемых на лучшие участки: 5;
- Количество пчел, отправляемых на другие выбранные участки: 2;
- Количество лучших участков: 2;
- Количество выбранных участков: 3;
- Размер области для каждого участка: 10;

Пусть пчелы-разведчики попали на следующие, участки (список отсортирован по убыванию целевой функции):

$$f(15, 18) = -549$$

$$f(-30, -15) = -1125$$

$$f(22, -31) = -1445$$

$$f(18, 40) = -1924$$

$$f(-25, 47) = -2834$$

$$f(60, 86) = -10996$$

$$f(-91, -99) = -18082$$

$$f(17, -136) = -18785$$

$$f(-152, -1) = -22501$$

$$f(-222, 157) = -73933$$

Согласно, установленным параметрам, выбираются 2 лучшие точки:

$$f(15, 18) = -549$$

$$f(-30, -15) = -1125$$

Затем определяются другие 3 перспективных участка:

$$f(22, -31) = -1445$$

$$f(18, 40) = -1924$$

$$f(-25, 47) = -2834$$

В окрестности лучших точек будут отправлены по 10 пчел:

Для первой лучшей точки значение координат, которыми ограничивается участок будет:

$$[15 - 10 = 5; 15 + 10 = 25] \text{ для первой координаты}$$

$[18 - 10 = 8; 18 + 10 = 28]$ для второй координаты

И для второй точки:

$[-30 - 10 = -40; -30 + 10 = -20]$ для первой координаты

$[-15 - 10 = -25; -15 + 10 = -5]$ для второй координаты

Аналогично рассчитываются интервалы для выбранных участков:

$[12; 32] [-41; -21]$

$[8; 28] [30; 50]$

$[-35; 15] [37; 57]$

Необходимо заметить, что по каждой из координат размер области одинаков и равен 20, в реальности это не обязательно так.

В каждый из лучших интервалов отправляем по 5 пчел, а на выбранные участки по 2 пчелы. Причем, мы не будем менять положение пчел, нашедших лучшие и выбранные участки, иначе есть вероятность того, что на следующей итерации максимальное значение целевой функции будет хуже, чем на предыдущем шаге.

Теперь пусть на первом лучшем участке имеются следующие пчелы:

$$f(15, 18) = -549$$

$$f(7, 12) = -193$$

$$f(10, 10) = -100$$

$$f(16, 24) = -832$$

$$f(18, 24) = -900$$

Как видно, из расчета, уже среди этих новых точек есть такие, которые лучше, чем предыдущее решение.

Так же поступаем и со вторым лучшим участком, а затем аналогично и с выбранными участками. После чего среди всех новых точек снова отмечаются лучшие и выбранные, а процесс повторяется заново [16].

3.4. Реализация классов программного кода

Рассмотрим пример реализации алгоритма роя пчел на языке Python. В примере используются две дополнительные библиотеки. Matplotlib используется для визуализации данных, а Psycopy для ускорения расчета.

Архив с исходниками содержит следующие файлы:

- ruBee.py – основные классы, реализующие алгоритм роя пчел.
- beetest.py – основной модуль примера, в котором содержатся все параметры алгоритма. Этот скрипт и нужно запускать.
- beeexamples.py – в этом модуле содержатся различные классы пчел для разных целевых функций.

- `beetestfunc.py` – вспомогательные функции для визуализации процесса расчета.

В модуле `pybee` содержатся следующие классы:

- `floatbee` – базовый класс для пчел, положение которых описывается числами с плавающей точкой.

- `hive` – класс улья, внутри которого и происходят основные действия алгоритма по выделению лучших и выбранных участков, а также отправка пчел на нужные позиции.

- `statistic` – вспомогательный класс для сбора статистики по сходимости алгоритма роя пчел. Этот класс накапливает лучшие результаты для каждого шага итерации.

Класс `floatbee`

Рассмотрим класс `floatbee`. Это базовый класс, от которого необходимо наследоваться и переопределить один метод, о котором поговорим чуть позже, кроме того в конструкторе нужно определить некоторые члены. А именно:

`self.position`

`self.minval`

`self.maxval`

Эти переменные должны быть типа списка (или массив) и содержать столько элементов, сколько аргументов имеет целевая функция, которую необходимо оптимизировать.

`self.position` хранит в себе одно решение, которое сопоставлено с данной пчелой. В конструкторе рекомендуется заполнять этот список случайными величинами.

`self.minval` и `self.maxval` хранят списки, содержащие соответственно минимальные и максимальные значения для координат из `self.position`.

Допустим, что положение у нас определяется двумя координатами, причем первая координата может изменяться в пределах `[-1; 1]`, а вторая в пределах `[-11; 12]`. Тогда заполнение этих членов может быть таким:

```
self.minval = [-1.0, -11.0]
```

```
self.maxval = [1.0, 12.0]
```

```
self.position = [random.uniform (self.minval[n], self.maxval[n]) for n in range (2) ]
```

Таким образом, размеры списков `self.position`, `self.minval` и `self.maxval` всегда должны совпадать.

После заполнения списков этих членов, необходимо перегрузить метод

```
def calcfitness (self)
```

Именно в нем рассчитывается целевая функция (или здоровье пчелы). Следует особо отметить, что само значение целевой функции не возвращается из этого метода, а присваивается внутри метода члену `self.fitness`. Сам метод не должен ничего возвращать.

Это сделано для того, чтобы целевая функция считалась только один раз при изменении координат пчелы. Причем метод `calcfitness()` должен быть обязательно выполнен после любых перемещений пчелы.

Рассмотрим пример.

Пусть целевая функция $f(x, y) = -x^2 - y^2$, тогда `calcfitness()` будет выглядеть следующим образом:

```
class mybee (pybee.floatbee):
```

```
...
```

```
def calcfitness (self):
```

```
    """Расчет целевой функции. Этот метод необходимо перегрузить в производном классе. Функция не возвращает значение целевой функции, а только устанавливает член self.fitness. Эту функцию необходимо вызывать после каждого изменения координат пчелы"""
```

```
    self.fitness = 0.0
```

```
    for val in self.position:
```

```
        self.fitness -= val * val
```

Тогда с учетом вышесказанного конструктор для пчелы будет выглядеть следующим образом:

```
class mybee (pybee.floatbee):
```

```
...
```

```
def __init__ (self):
```

```
    self.minval = [-1.0, -11.0]
```

```
    self.maxval = [1.0, 12.0]
```

```
    self.position = [random.uniform (self.minval[n], self.maxval[n]) for n in range (2) ]
```

```
    self.calcfitness()
```

С точки зрения пользователя больше про класс пчел ничего знать не обязательно. Оставшиеся члены класса `floatbee` будут кратко рассмотрены по мере их упоминания при разборе основного алгоритма.

Класс `hive`

Основные шаги алгоритма роя пчел реализованы внутри класса hive (улей).

Рассмотрим параметры конструктора

class hive:

...

```
def __init__(self, scoutbeecount, selectedbeecount, bestbeecount, \
             selsitescount, bestsitescount, \
             range_list, beetype):
```

Как видно, в конструктор передается 7 параметров (не считая self):

- scoutbeecount – количество пчел-разведчиков.
- selectedbeecount – количество пчел, посылаемых на каждый из выбранных (не лучших) участков. Имеется в виду, сколько пчел будет отправлено на один участок.
- bestbeecount – количество пчел, посылаемых на каждый из лучших участков.
- selsitescount – количество выбранных (перспективных, но не лучших) участков.
- bestsitescount – количество лучших участков.
- range_list – список, элементы которого определяют размеры выбранных и лучших участков по каждой координате
- beetype – класс пчел, производный от floatbee, который используется в алгоритме.

С первыми пятью параметрами и последним все понятно, рассмотрим параметр range_list, который представляет собой список размеров областей, куда посылаются пчелы. Его роль в алгоритме проще показать на примере.

Пусть у нас положение участков определяется двумя координатами (целевая функция имеет два параметра, или пчелы "живут" в двумерном мире), тогда список (или массив) range_list должен иметь два элемента. Пусть это будут [1.0, 2.0]. Это значит, что при отправке пчелы в область какой-то точки (x; y), значение ее первой координаты будет случайным образом выбрано из интервала [x - 1.0; x + 1.0], а значение второй координаты - [y - 2.0; y + 2.0]. То есть, если мы отправим пчелу в область точки (10.0, 20.0), то в реальности получим координаты, которые будут лежать в пределах от 9.0 до 11.0 для первой координаты и от 18.0 до 22.0 для второй.

Кстати, для отправки пчелы в окрестность точки в классе floatbee есть метод

```
def goto (self, otherpos, range_list)
```


В котором otherpos – точка, в окрестность которой посылается пчела, а range_list – тот же самый список размеров окрестностей,

Возвращаемся к конструктору класса hive. Рассмотрим его код class hive:

...

```
def __init__(self, scoutbeecount, selectedbeecount, bestbeecount, \
             selsitescount, bestsitescount, \
             range_list, beetype):

    self.scoutbeecount = scoutbeecount
    self.selectedbeecount = selectedbeecount
    self.bestbeecount = bestbeecount

    self.selsitescount = selsitescount
    self.bestsitescount = bestsitescount

    self.beetype = beetype

    self.range = range_list

    # Лучшая на данный момент позиция
    self.bestposition = None

    # Лучшее на данный момент здоровье пчелы
    self.bestfitness = -1.0e9

    # Начальное заполнение роя пчелами со случайными координатами
    beecount = scoutbeecount + selectedbeecount * selsitescount +
    bestbeecount * bestsitescount
    self.swarm = [beetype() for n in xrange (beecount)]

    # Лучшие и выбранные места
    self.bestsites = []
    self.selsites = []

    self.swarm.sort (floatbee.sort, reverse = True)
```

```
self.bestposition = self.swarm[0].getposition()
self.bestfitness = self.swarm[0].fitness
```

В нем сохраняются все передаваемые параметры и создается член `self.bestposition`, который хранит в себе лучшее на данный момент решение.

Значение целевой функции для лучшего на данный момент решения хранится в члене `self.bestfitness`.

Затем создается рой пчел. Общее количество пчел в нем равно

```
beecount = scoutbeecount + selectedbeecount * selsitescount + bestbeecount *
bestsitescount
```

То есть сумма количества разведчиков, количества пчел посылаемых на один выбранный участок, умноженное на количество выбранных участков, и количества пчел, посылаемых на один из лучших участков, умноженное на количество лучших участков.

Затем заполняется список члена `self.swarm` (рой):

```
self.swarm = [beetype() for n in xrange (beecount)]
```

В конструкторе все параметры пчелы должны инициализироваться случайным образом, поэтому получается, что при создании роя пчел, все они выступают в роли пчел-разведчиков, которых отправляют на случайные точки. С точки зрения алгоритма это не обязательно должно быть так. Можно в первый раз отправить только заданное количество пчел-разведчиков, а пчел, предназначенных для лучших и выбранных участков, отправлять на следующих итерациях. Но в данном случае в самом начале на случайные места отправляются все пчелы, участвующие в алгоритме.

После того как рой создан, он сортируется по убыванию целевой функции у каждой пчелы. В качестве функции сравнения передается метод `sort()` из класса `floatbee`.

После сортировки лучшим будет решение у пчелы, находящейся в самом начале списка. Значение целевой функции для данной пчелы сохраняется в член `self.bestfitness`, а в член `self.bestposition` сохраняется копия координат лучшей пчелы. Для получения копии координат используется метод `getposition()` из класса `floatbee`:

```
class floatbee:
```

```
...
```

```
def getposition (self):
    """Вернуть копию своих координат"""
    return [val for val in self.position]
```

Для выполнения одной итерации алгоритма роя пчел необходимо из экземпляра класса `hive` вызвать метод

```
def nextstep (self)
```

Этот метод состоит из двух логических этапов. Сначала выбираются лучшие и выбранные участки. Выбор лучших участков происходит следующим образом:

```
class hive:
```

```
...
```

```
def nextstep (self):
```

```
    # Выбираем самые лучшие места и сохраняем ссылки на тех, кто их нашел
```

```
    self.bestsites = [ self.swarm[0] ]
```

```
    curr_index = 1
```

```
    for currbee in self.swarm [curr_index: -1]:
```

```
        # Если пчела находится в пределах уже отмеченного лучшего участка, то ее положение не считаем
```

```
        if currbee.otherpatch (self.bestsites, self.range):  
            self.bestsites.append (currbee)
```

```
        if len (self.bestsites) == self.bestsitescount:  
            break
```

```
    curr_index += 1
```

```
...
```

Пчелы, нашедшие лучшие участки помещаются в список `self.bestsites`. В первых, туда помещается пчела, находящаяся в начале списка, а затем происходит перебор остальных пчел до тех пор, пока не будет отобрано нужное количество. Причем пропускаются пчелы, которые находятся в одном участке с пчелой, которая уже находится в списке `self.bestsites`. `curr_index` хранит номер пчелы в списке, которую будем проверять следующей.

Проверка того является ли участок, найденный пчелой новым или какая-то пчела уже находится поблизости, происходит внутри метода `otherpatch()` класса `floatbee`.

```
class floatbee:
```

...

```
def otherpatch (self, bee_list, range_list):
    """Проверить находится ли пчела на том же участке, что и одна из пчел в bee_list.
    range_list - интервал изменения каждой из координат"""
    if len (bee_list) == 0:
        return True

    for curr_bee in bee_list:
        position = curr_bee.getposition()

        for n in xrange ( len (self.position) ):
            if abs (self.position[n] - position[n]) > range_list[n]:
                return True

    return False
```

В данном случае в качестве bee_list передается список лучших пчел self.bestsites, а в качестве range_list все тот же список размеров окрестностей. Участок, найденный пчелой, считается новым, если в списке bee_list нет пчелы, расстояние до которой по каждой из координат не превышает соответствующего размера из range_list.

После нахождения таким образом лучших участков аналогично отмечаем выбранные участки.

```
class hive:
```

...

```
def nextstep (self):
    ...

    self.selsites = []

    for currbee in self.swarm [curr_index: -1]:
        if currbee.otherpatch (self.bestsites, self.range) and currbee.otherpatch (self.selsites, self.range):
            self.selsites.append (currbee)
```

```

        if len (self.selsites) == self.selsitescount:
            break

```

Пчелы с выбранных участков сохраняются в список self.selsites.

После того как отмечены лучшие и выбранные участки, отправим пчел на соответствующие позиции.

```

class hive:

```

```

    ...

```

```

    def nextstep (self):

```

```

        ...

```

```

        # Номер очередной отправляемой пчелы. 0-ую пчелу никуда не отправляем

```

```

        bee_index = 1

```

```

        for best_bee in self.bestsites:

```

```

            bee_index = self.sendbees (best_bee.getposition(), bee_index, self.
            bestbeecount)

```

```

        for sel_bee in self.selsites:

```

```

            bee_index = self.sendbees (sel_bee.getposition(), bee_index, self.s
            electedbeecount)

```

```

        # Оставшихся пчел пошлем куда попадет

```

```

        for curr_bee in self.swarm[bee_index: -1]:

```

```

            curr_bee.gotorandom()

```

```

        self.swarm.sort (floatbee.sort, reverse = True)

```

```

        self.bestposition = self.swarm[0].getposition()

```

```

        self.bestfitness = self.swarm[0].fitness

```

За отправку пчел отвечает метод sendbees() класса hive. Суть состоит в том, что она отправляет заданное количество пчел, пропуская тех, которые уже находятся среди лучших и выбранных пчел. Внутри sendbees() вызывается метод goto() класса floatbee. Его мы рассмотрим подробнее.

```

class floatbee:

```

```

    ...

```

```
def goto (self, otherpos, range_list):
    """Перелет в окрестность места, которое нашла другая пчела. Не в то же
самое место! """
```

```
    # К каждой из координат добавляем случайное значение
    self.position = [otherpos[n] + random.uniform (-range_list[n], range_list[n]) \
        for n in xrange (len (otherpos) ) ]
```

```
    # Проверим, чтобы не выйти за заданные пределы
    self.checkposition()
```

```
    # Расчитаем и сохраним целевую функцию
    self.calcfitness()
```

Сначала заполняется список новых координат, которые складываются из точки, в которую отправляют пчелу и случайной величины в интервале от -range_list[n] до range_list[n].

Затем вызывается метод checkposition(), который корректирует координаты, если они выходят за заданные пределы.

```
class floatbee:
```

```
    ...
```

```
    def checkposition (self):
        """Скорректировать координаты пчелы, если они выходят за установленные пределы"""
        for n in range ( len (self.position) ):
            if self.position[n] < self.minval[n]:
                self.position[n] = self.minval[n]

            elif self.position[n] > self.maxval[n]:
                self.position[n] = self.maxval[n]
```

После того как были отправлены пчелы на лучшие и выбранные места, оставшихся пчел из улья отправляются на случайные координаты. Затем все пчелы опять сортируются по убыванию целевой функции и сохраняются лучшие позиции и лучшее значение целевой функции.

Метод nextstep() вызывается пользователем до тех пор пока не будет выполнено одно из условий останова, но об этом мы поговорим, когда будем разбирать примеры использования алгоритма роя пчел.

Класс statistic

В модуле `pybme` есть еще один класс, который не используется в алгоритме непосредственно, но который может помочь в отладке и визуализации результатов. Это класс `statistic`, который сохраняет найденное решение для каждого шага итерации. Причем, один экземпляр класса `statistic` можно использовать для нескольких независимых запусков алгоритма. Это может понадобиться для того, чтобы посмотреть, например, как сходятся параметры при усреднении по нескольким запускам.

Рассмотрим конструктор класса `statistic` и обсудим способ хранения данных в нем.

`class statistic:`

```
""" Класс для сбора статистики по запускам алгоритма """
def __init__(self):
    # Индекс каждого списка соответствует итерации.
    # В элементе каждого списка хранится список значений для каж-
    дого запуска
    # Добавлять надо каждую итерацию

    # Значения целевой функции в зависимости от номера итерации
    self.fitness = []

    # Значения координат в зависимости от итерации
    self.positions = []

    # Размеры областей в зависимости от итерации
    self.range = []
```

В классе содержатся три списка, которые хранят значения целевой функции, координат и размеров областей. Хранятся значения следующим образом, `self.positions` хранит список, первый индекс которого обозначает номер запуска алгоритма (начиная с 0). Внутри каждого элемента хранится список, каждый элемент которого обозначает номер итерации алгоритма. Внутри этого второго списка хранится список координат, лучших на данную итерацию. То есть, чтобы получить k -ую координату, которая была при m -ой итерации при n -ом запуске алгоритма, необходимо написать следующее выражение (считаем, что `stat` - экземпляр класса `statistic`):

```
stat.positions[n][m][k]
```

Аналогично со списком `range`. Список `fitness` немного проще из-за того, что целевая функция представляет собой число, а не список, то есть индекса k у нее нет.

Для каждой итерации алгоритма необходимо вызывать метод `add()`, который добавляет результат в статистику:

```
def add (self, runnumber, currhive)
```

В нем сохраняются в соответствующих списках значения целевой функции, координаты и интервал изменения диапазонов координат согласно структуре, которую была рассмотрена.

3.5. Пример работы программного кода алгоритма

В качестве примеров рассмотрим только класс пчел, производный от `floatbee`, остальные классы будут устроены аналогичным образом. Все классы пчел для различных целевых функций описаны в файле `beexamples.py`.

Итак, пусть наша целевая функция имеет вид (так называемая гиперсфера)

$$f(x_1, \dots, x_n) = - \sum_{i=0}^n x_i^2$$

Максимум такой функции находится в точке, когда все координаты равны 0. Рассмотрим класс пчелы для этой целевой функции.

```
class spherebee (pybee.floatbee):
```

```
    """Функция - сумма квадратов по каждой координате"""
```

```
    # Количество координат
```

```
    count = 20
```

```
    def __init__ (self):
```

```
        pybee.floatbee.__init__ (self)
```

```
        self.minval = [-150.0] * spherebee.count
```

```
        self.maxval = [150.0] * spherebee.count
```

```
        self.position = [random.uniform (self.minval[n], self.maxval[n]) for n in
                           xrange (spherebee.count) ]
```

```
        self.calcfitness()
```

```
    def calcfitness (self):
```



```
"""Расчет целевой функции. Этот метод необходимо перегрузить в
производном классе. Функция не возвращает значение целевой
функции, а только устанавливает член self.fitness. Эту функцию
необходимо вызывать после каждого изменения координат пче-
лы"""
```

```
self.fitness = 0.0
for val in self.position:
    self.fitness -= val * val
```

```
@staticmethod
def getstartrange ():
    return [150.0] * spherebee.count
```

```
@staticmethod
def getrangekoeff ():
    return [0.98] * spherebee.count
```

Обратите внимание на переменную count внутри класса, она обозначает количество координат (n) гиперсферы, в других классах пчел так же используется эта переменная.

Рассмотрим конструктора. В нем для порядка сначала вызывается конструктор базового класса floatbee, затем создаются три списка:

- список минимальных значений для каждой из координат (self.minval).
- список максимальных значений (self.maxval).
- список текущих координат (self.position), заполненный случайными величинами.

После заполнения координатами списка self.position вызывается метод self.calcfitness() для вычисления целевой функции, используя текущие координаты.

Затем реализуются два очень простых статических метода, которые облегчают запуск примеров независимо от того какой класс пчел используется. Метод getstartrange() используется для задания начальных размеров для каждой из областей. Именно этот список и передается в конструктор класса hive.

Кроме того статический метод getrangekoeff(), возвращает список коэффициентов, определяющих во сколько раз будут уменьшаться размеры областей при необходимости. В данном примере эти коэффициенты равны 0.98, т.е.

каждый раз, когда надо будет сузить область поиска, элементы списка `range` класса `hive` будут умножаться на 0.98.

Теперь рассмотрим скрипт `beetest.py`, из которого и происходит запуск алгоритма. Этот скрипт требует, чтобы был установлен пакет `pylab`, который используется для визуализации данных. Кроме того, очень желательно, чтобы был установлен пакет `psyco`, который существенно ускоряет расчет, хотя и без него скрипт будет работать.

Думаю, что нет смысла описывать очень подробно каждую строку из `beetest.py`, рассмотрим только ключевые моменты. Сначала создается класс для сбора статистики

```
stat = pybee.statistic()
```

Параметры алгоритма роя пчел задаются после комментария

```
#####
```

Параметры алгоритма

```
#####
```

В первую очередь задается класс используемых пчел.

```
beetype = beeexamples.spherebee
```

```
#beetype = beeexamples.dejongbee
```

```
#beetype = beeexamples.goldsteinbee
```

```
#beetype = beeexamples.rosenbrockbee
```

```
#beetype = beeexamples.testbee
```

```
#beetype = beeexamples.funcbee
```

Затем идет задание других параметров алгоритма:

```
# Количество пчел-разведчиков
```

```
scoutbeecount = 300
```

```
# Количество пчел, отправляемых на выбранные, но не лучшие участки
```

```
selectedbeecount = 10
```

```
# Количество пчел, отправляемые на лучшие участки
```

```
bestbeecount = 30
```

```
# Количество выбранных, но не лучших, участков
```

```
selsitescount = 15
```

```
# Количество лучших участков
```

```
bestsitescount = 5
```

Далее располагаются другие параметры.

```
# Количество запусков алгоритма
```

```
runcount = 1
```

```
# Максимальное количество итераций
```

```
maxiteration = 1000
```

```
# Через такое количество итераций без нахождения лучшего решения умень-  
шим область поиска
```

```
max_func_counter = 10
```

Рассмотрим их более подробно;

- `runcount` – это количество независимых запусков алгоритма. Так как алгоритм во многом зависит от случайных величин, то многократный запуск, а также усреднение по всем запускам будет более наглядно показывать сходимость алгоритма, чем запуск только один раз. Хотя в данном случае по умолчанию алгоритм запускается именно один раз.

- `maxiteration` - это максимальное количество итераций алгоритма. В данном случае это единственный критерий останова. Это сделано для того, чтобы было проще усреднять результаты работы по нескольким запускам. Если бы был введен еще другой критерий останова, то в классе для сбора статистики для каждого запуска списки с результатами были бы разной длины.

- `max_func_counter` определяет через сколько итераций, которые не дают лучшего решения, будут уменьшаться размеры областей для поиска решения. То есть в данном случае, если в течение 10 итераций не будет найдено решение лучше, чем было до этого, то область, задаваемая списком `hive.range`, будет уменьшена в соответствии со списком, возвращаемым статическим методом `gettrangecoeff()` класса пчелы, то есть в данном случае размер будет умножен на 0.98.

Рассмотрим еще некоторые участки кода внутри цикла по количеству запусков алгоритма. Создается класс улей:

```
currhive = pybee.hive (scoutbeecount, selectedbeecount, bestbeecount, \  
selsitescount, bestsitescount, \  
beetype.getstartrange(), beetype)
```

Добавляем текущее лучшее решение в статистику

```
stat.add (runnumber, currhive)
```

Создаются необходимые переменные для подсчета количества неудачных итераций (когда более хорошее решение не было найдено)

```
# Начальное значение целевой функции
```

```
best_func = -1.0e9
```

```
# Количество итераций без улучшения целевой функции
```

```
func_counter = 0
```

Рассмотрим основной цикл по всем итерациям.

```

for n in xrange (maxiteration):
    currhive.nextstep ()

    stat.add (runnumber, currhive)

    if currhive.bestfitness != best_func:
        # Найдено место, где целевая функция лучше
        best_func = currhive.bestfitness
        func_counter = 0

        # Обновим рисунок роя пчел
        #plotswarm (currhive, 0, 1)

        print "\n*** iteration %d / %d" % (runnumber + 1, n)
        print "Best position: %s" % (str (currhive.bestposition) )
        print "Best fitness: %f" % currhive.bestfitness
    else:
        func_counter += 1
        if func_counter == max_func_counter:
            # Уменьшим размеры участков
            currhive.range = [currhive.range[m] *
                               koeff[m] for m in xrange ( len (currhive.range) ) ]
            func_counter = 0

            print "\n*** iteration %d / %d (new range)" % (runnumber + 1, n)
            print "New range: %s" % (str (currhive.range) )
            print "Best position: %s" % (str (currhive.bestposition) )
            print "Best fitness: %f" % currhive.bestfitness

#if n % 10 == 0:
    #plotswarm (currhive, 2, 3)

```

Здесь n соответствует номеру итерации (начиная с 0). Сначала вызывается метод nextstep() для новой итерации, а затем результат добавляется в статистику. Потом, если на данной итерации было найдено лучшее решение, то оно сохраняется, а в консоль выводится номер итерации, новое найденное решение и значение целевой функции.

Если на данной итерации новое решение не найдено, то увеличивается счетчик `func_counter` для неудачных итераций, и если его значение стало равно `max_func_counter`, то происходит сужение областей для каждой из координат. После чего, опять же, выводится в консоль номер итерации, новые диапазоны, лучшие на данный момент решение и значение целевой функции.

Все остальные строки исходника, в том числе, идущие и после цикла по итерациям, предназначены для наглядного представления результатов работы алгоритма.

Все функции для визуализации с помощью библиотеки `pylab` находятся в файле `beetestfunc.py`.

Во-первых, это функция `plotswarm()`, которая на плоскости отмечает расположение каждой пчелы. В качестве первого параметра в нее необходимо передать экземпляр класса `hive`. Вторым и третьим параметрами – номера индексов координат пчелы, отображаемые на плоскости. Так как пчелы располагаются на плоскости, а у целевой функции может быть больше параметров (координат пчелы), то можно отобразить только две координаты, которые и задаются с помощью этих двух параметров.

По умолчанию отображение пчел отключено для ускорения счета, но его можно включить, причем обновление рисунка может происходить в двух случаях: если было найдено более хорошее решение, и может быть обновление на каждой 10-ой итерации. Для первого случая необходимо раскомментировать строку.

```
#beetestfunc.plotswarm (currhive, 0, 1)
```

После комментария.

```
# Обновим рисунок роя пчел
```

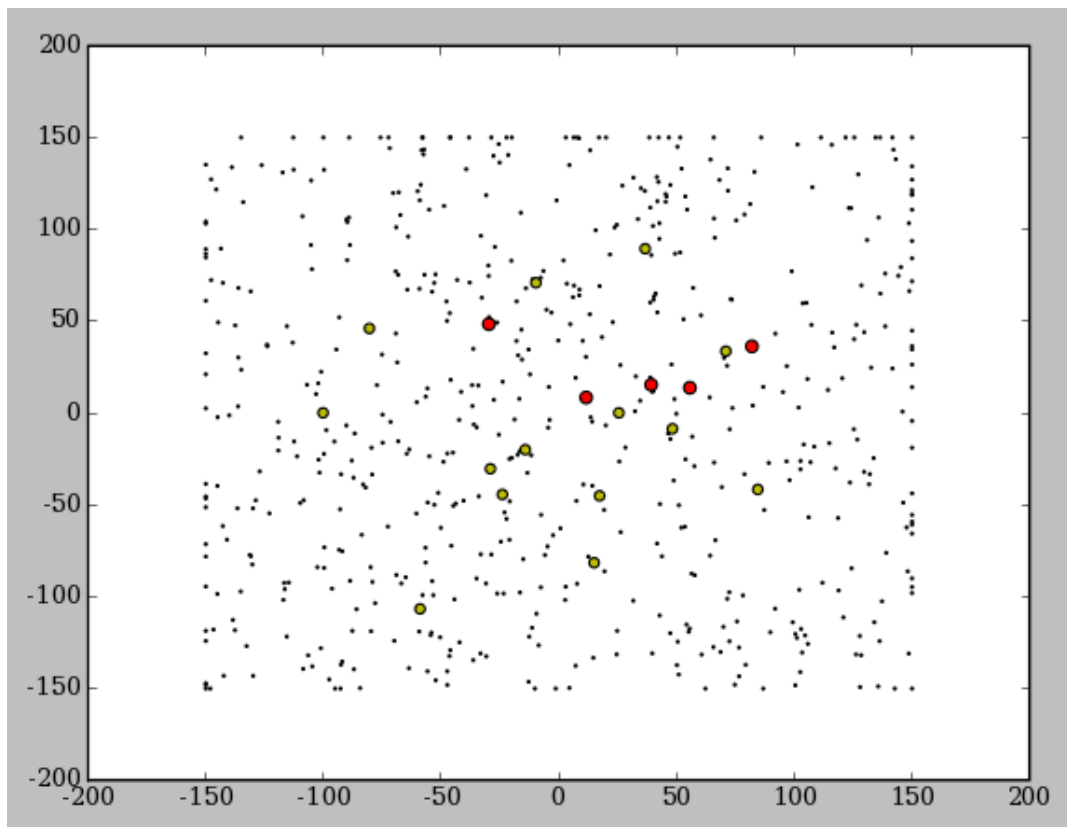
Во втором случае необходимо раскомментировать две строки

```
#if n % 10 == 0:
```

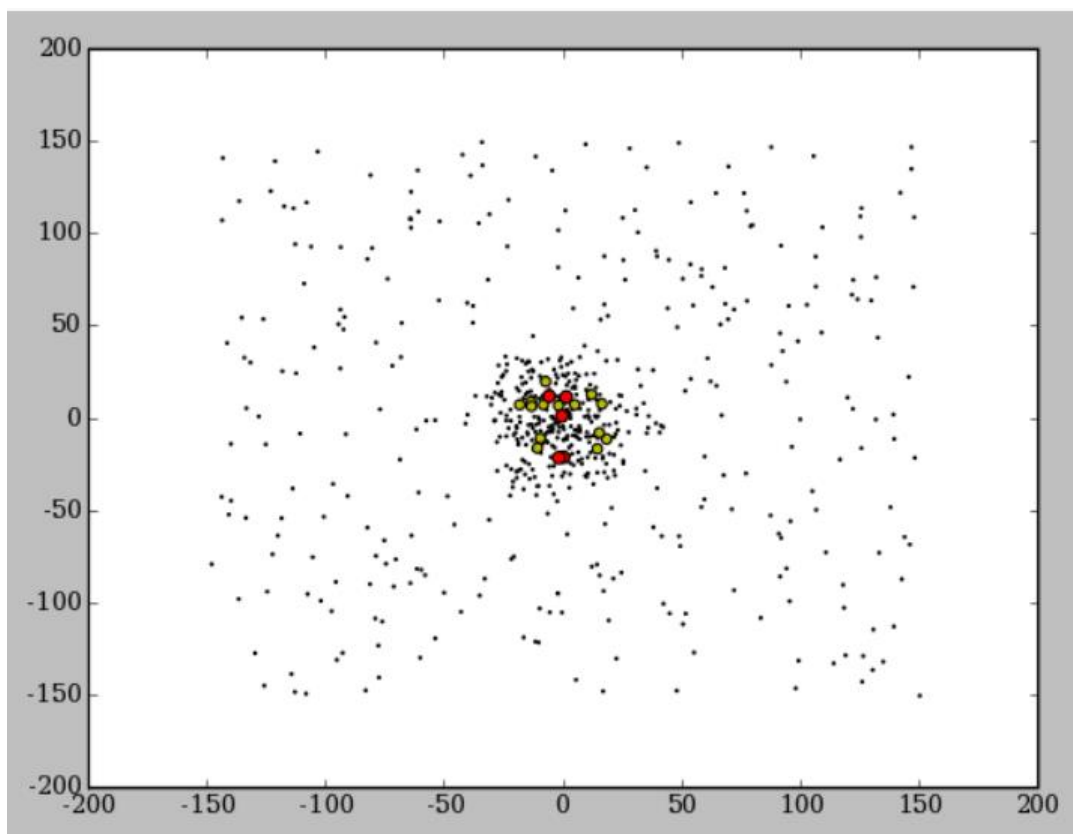
```
#beetestfunc.plotswarm (currhive, 2, 3)
```

Отметим, что для простоты и наглядности скрипта отображение роя пчел происходит в том же потоке, что и выполнение основного алгоритма, поэтому обновление картинки может "виснуть", если переключиться на другое приложение, при этом сам алгоритм будет работать.

На рис.3.3 видно, как пчелы постепенно скапливаются вокруг одного лучшего решения. На этом рисунке красные точки - пчелы, нашедшие лучшие решения, желтые точки - выбранные решения, а черные точки - остальные пчелы, включая пчел-разведчиков, которые располагаются случайным образом.



а) Начало работы пчелиного алгоритма



б) Скопление пчел вокруг одного решения

Рис. 3.3. Визуализация работы пчелиного алгоритма

Также в конце расчета выводятся графики сходимости решения и значения целевой функции в зависимости от номера итерации.

4. ОПИСАНИЕ МАЛОИЗВЕСТНЫХ РОЕВЫХ АЛГОРИТМОВ

В последнее время появляется все большее число роевых алгоритмов оптимизации, вдохновленных поведением живых существ в природе, а также инспирированных неживой природой. В разных зарубежных публикациях их называют не только роевыми, но и поведенческими, интеллектуальными, метаэвристическими, многоагентными, популяционными [17].

4.1. Алгоритм роя светлячков

Существует около двух тысяч видов светлячков, которые способны светиться, производя ритмичные и короткие вспышки. Изучая поведение светлячков в естественной среде обитания, было замечено, что структура вспышек у каждого вида уникальна. Свечение у светлячков служит для коммуникации между особями. Каждая особь выделяет люциферин (особый пигмент, способствующий свечению), от количества которого зависит интенсивность свечения. Они используются для привлечения особей противоположного пола, а также для привлечения потенциальной добычи. Кроме того, вспышки света могут выступать в качестве защитного механизма.

Алгоритм светлячков предложен в 2007 г. Янгом (X.-Sh. Yang). Модель поведения светлячков представлена следующим образом: все виды светлячков имеют возможность привлекать друг друга вне зависимости от пола; чем больше яркость светлячка, тем привлекательнее он для других особей; светлячки, с меньшей степенью привлекательности, притягиваются к более привлекательным особям; яркость излучения каждого светлячка, видимая другим светлячком, сокращается при увеличении расстояния между особями; если в окружении светлячка нет более ярких особей, то он перемещается произвольным образом [18].

Для формального описания модели используем вместо понятия «светлячок» понятие «агент». При инициализации поиска все агенты произвольным образом распределены в поисковом пространстве целевой функции. Каждый агент выделяет определенное количество люциферина и имеет свою собственную область принятия решений. Агент i рассматривает другого агента j как соседа, если он находится в пределах радиуса окрестности поиска агента i и уровень люциферина агента j выше, чем агента i , т.е. $I_j > I_i$. Локальная область принятия решений задается радиусом окрестности поиска для каждого i -го агента. Используя вероятностный механизм, каждый агент выбирает соседнего агента, у которого уровень люциферина выше, чем его собственный, и движется в его направлении. Иными словами, каждый агент движется в направлении

того агента, у которого уровень свечения выше. Интенсивность свечения каждого агента определяется значением целевой функции в текущем положении. Чем выше интенсивность свечения, тем больше значение целевой функции. Кроме того, радиус окрестности поиска каждого агента зависит от количества агентов в этой области. Если в окрестности поиска находится малое количество агентов, то ее радиус увеличивается. В противном случае, радиус окрестности поиска сокращается. Иными словами, данный алгоритм имеет 4 глобальных этапа: начальное распределение агентов в пространстве поиска, обновление уровня люциферина, перемещение агентов в более перспективную область поиска, обновление радиуса окрестности поиска каждого агента (рис. 4.1) [17].

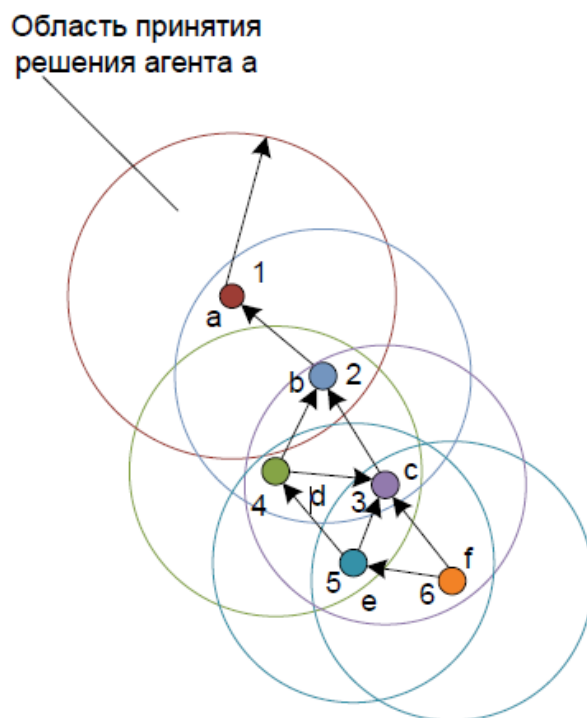


Рис. 4.1. Пример перемещения агента в пространстве поиска

На рис. 4.1 приведен направленный граф на 6 вершин, отражающий зависимость движения каждого агента от уровня люциферина, от которого зависит размер локальной области принятия решения. Агенты ранжируются в порядке возрастания уровня люциферина. В данном случае у агентов a, b, c и d уровень люциферина больше, чем у агента e , который расположен в пределах области локального принятия решений только агентов c и d , т.е. агент e имеет только два направления для перемещения.

Представим алгоритм, инспирированного поведением роя светлячков [19]:

Шаг 1. Инициализация входных параметров: параметра β для определения радиуса окрестности ($0 < \beta < 1$); параметра ρ определения уровня люциферина ($0 < \rho < 1$); параметра δ для генерации новой позиции ($0 < \delta < 1$);

начального радиуса r_0 окрестности; максимального числа итераций n ; размера популяции k ; длины вектора позиции агента m ; минимальных x_{min} и максимальных x_{max} значений для вектора позиций.

Шаг 2. Задание функции цели $F(x) \rightarrow \min$, где x – вектор позиции жука-светляка.

Шаг 3. Создание случайным образом вектора лучшей позиции

$$x = (x_1, \dots, x_m), x_j = x_j^{min} + (x_j^{max} - x_j^{min})rand(),$$

где $rand()$ – функция возвращающая равномерно распределённое случайное число в диапазоне $[0,1]$.

Шаг 4. Размещение в поисковом пространстве исходной популяции P ;

Шаг 5. Создание случайным образом вектора позиции x_k .

$$x_k = (x_{k1}, \dots, x_{km}), x_{kj} = x_j^{min} + (x_j^{max} - x_j^{min})rand(),$$

Шаг 6. Изначально, все агенты имеют одинаковое количество люциферина. Инициализация количества люциферина l_k .

$$l_k = l_0$$

Инициализация радиуса окрестности r_k .

$$r_k = r_0$$

Шаг 7. Обновление уровня люциферина зависит от позиции агента в пространстве (значения его целевой функции). Все агенты на начальной итерации имеют одинаковый уровень люциферина, поэтому значение целевой функции каждого агента зависит от его положения в пространстве поиска. Уровень люциферина каждого агента увеличивается пропорционально измеряемым характеристикам агента (температура, уровень излучения). С точки зрения оптимизации это и является целевой функцией. Для моделирования процесса распада флуоресцирующего вещества производится вычитание части люциферина.

Вычисление уровня люциферина $l_k(t + 1)$ (уровня свечения) i -го агента в момент времени t

$$l_k(t + 1) = (1 - \rho)l_k(t) + \gamma F(x_k)(t + 1), k \notin \overline{1, K}$$

где ρ – коэффициент ослабления уровня люциферина для моделирования процесса распада флуоресцирующего вещества, γ – коэффициент привлекательности светлячка, $F(x_k)(t + 1)$ – значение целевой функции k -го агента в момент времени $t + 1$.

Шаг 8. Миграция жуков-агентов (перемещение). Каждый агент выбирает того агента внутри радиуса окрестности поиска, у которого уровень люциферина выше, чем его собственный. Задание $N_i(t)$ – множества соседей i -го агента в момент времени t , r_k – радиуса окрестности поиска k -го светлячка в момент времени t .

Создание окрестности для k -го жука-светляка.

$$U_k = \{m \mid |x_m - x_k| < r_k, l_k < l_m, m \in \overline{1, K}\}$$

Шаг 9. При обновлении своего положения в пространстве поиска каждый агент на основе вероятностного механизма передвигается в направлении того агента, у которого уровень люциферина выше, чем его собственный (рис. 4.1).

Вычисление вероятности движения одного жука к другому.

$$P_{km} = \frac{l_m - l_k}{\sum_{s \in U_k} (l_s - l_k)}, m \in \overline{1, K}$$

Шаг 10. Выбор номера жука-светляка.

$$\text{Если } \sum_{s=1}^{c-1} P_{ks} < \text{rand}() \leq \sum_{s=1}^c P_{ks}, \text{ то } m = c.$$

Агент k , используя метод колеса рулетки, выбирает агента m и перемещается в его направлении.

Шаг 11. Модификация позиции. Определяется обновленная позиция агента k по формуле

$$x_k(t+1) = x_k(t) + \delta \frac{x_m(t) - x_k(t)}{\|x_m(t) - x_k(t)\|}$$

Шаг 11. Модификация радиуса окрестности. Обновление радиуса окрестности r_k поиска по формуле

$$r_k = \min\{r_{\max}, \max\{r_{\min}, r_k + \beta(n_U - |U_k|)\}\},$$

где $|U_k|$ – размер текущей окрестности; n_U – параметр для управления количеством соседних агентов.

Шаг 11. Определение лучшего жука-светляка по функции цели.

$$k^* = \operatorname{argmin} F(x_k).$$

Если $F(x_k) < F(x^*)$, то $x^* = x_k$. Вывод результата x^* .

Соответственно, величины $\rho, \gamma, \delta, \beta, n_U$ – параметры алгоритма, значение которых определяется экспериментальным путем [19].

К положительным сторонам алгоритма можно отнести высокую точность получаемых результатов. К отрицательным качествам относится длительное время работы алгоритма при совсем не критичных значениях количества итераций и размером популяций. Наиболее важной особенностью алгоритма является зависимость между люциферинном и радиусом окрестности скопления агентов. Эта зависимость проявляется в том, что при увеличении люциферина уменьшается радиус поиска и при этом увеличивается точность работы алгоритма.

4.2. Алгоритм кукушкиного поиска

Алгоритм кукушкиного поиска предложили Янг (X.-Sh. Yang) и Деб (S. Deb) в 2009 г. Алгоритм вдохновлен поведением кукушек в процессе вынужденного гнездового паразитизма [20].

Различные виды кукушек откладывают яйца в коллективные гнезда вместе с другими кукушками, хотя могут выбрасывать яйца конкурентов, чтобы увеличить вероятность вылупления их собственных птенцов. Целый ряд видов кукушек занимается гнездовым паразитизмом, подкладывая свои яйца в гнезда других птиц как своего вида, так и, часто, других видов. Некоторые птицы могут конфликтовать с вторжением кукушек. К примеру, если хозяин гнезда обнаружит в нем яйца иного вида, то он либо выбросит эти яйца, либо просто покинет данное гнездо и соорудит на новом месте другое гнездо.

В алгоритме кукушкиного поиска каждое яйцо в гнезде представляет собой решение, а яйцо кукушки – новое решение. Цель заключается в использовании новых и потенциально лучших (кукушкиных) решений, чтобы заменить менее хорошие решения в гнёздах. В простейшем варианте алгоритма в каждом гнезде находится по одному яйцу. Положим, что речь идет о задаче глобальной безусловной максимизации. Алгоритм основан на следующих правилах: каждая кукушка откладывает одно яйцо за один раз в случайно выбранное гнездо; лучшие гнёзда с яйцами высокого качества (высоким значением пригодности) переходят в следующее поколение; яйцо кукушки, отложенное в гнездо, может быть обнаружено хозяином с некоторой вероятностью $\varepsilon_a(0; 1)$ и удалено из гнезда[21].

Схему алгоритма кукушкиного поиска можно представить в следующем виде [22]:

Шаг 1. Инициализируем популяцию $S = (s_i, i \in [1: |S|])$ из $|S|$ хозяйских гнезд и кукушку, т. е. определяем начальные значения компонентов векторов $X_i; i \in [1: |S|]$ и вектор начального положения кукушки X_c ;

Шаг 2. Выполняем некоторое число случайных перемещений кукушки в пространстве поиска с помощью полётов Леви и находим новое положение кукушки X_c .

Полет Леви – это движение, состоящее из серий коротких перемещений, причем в промежутках между ними совершаются длинные перемещения. Если прочертить траекторию такого движения, то получится большая фигура, состоящая из маленьких, которые по форме напоминают большую. Полеты Леви имеют отношение к фракталам, так как в них фрагменты являются подобием целого, иначе говоря, это самоподобные структуры. Называются они так, потому что теоретическую базу для их открытия создал французский математик

Поль Леви. Таким образом движутся Галактики, космическая пыль, пузырьки газа в жидкости и многие другие природные объекты.

Шаг 3. Случайным образом выбираем гнездо s_i , $i \in [1:|S|]$, и, если $\varphi(X_c) > \varphi(X_i)$, то заменяем яйцо в этом гнезде на яйцо кукушки, т. е. полагаем $X_i = X_c$;

Шаг 4. С вероятностью ε_a удаляем из популяции некоторое число худших случайно выбранных гнёзд (включая, быть может, гнездо s_i) и по правилам шага 1 строим такое же число новых гнёзд;

Шаг 5. Если условие окончания итераций не выполнено, то переходим к шагу 2.

Основные этапы поиска кукушки можно представить в виде псевдокода следующим образом:

begin

Генерация начальной популяции n гнезд x_j , ($j = 1, 2, \dots, n$);

while (критерий останова)

 Случайным образом поместить кукушку в точку x_i , выполняя полёты Леви;

 Случайным образом выбрать гнездо j среди n гнезд;

 if ($F_i < F_j$)

 Заменить x_j на новое решение x_i ;

 end if

 Часть гнезд, обнаруженных с вероятностью ε_a , удалить из популяции и построить такое же число новых;

 Сохранить лучшее решение(гнездо);

end while

Пост-обработка результатов и визуализация;

end

Полёты Леви кукушки реализуем по формуле:

$$X_c' = X_c + V \otimes L|X|(\lambda),$$

где обычно полагают все компоненты вектора V одинаковыми и равными v : $V = (v_j = v, j \in [1:|X|])$. Величина v должна быть связана с масштабами области поиска.

Уравнение представляет собой стохастическое уравнение случайных блужданий. В общем, случайное блуждание представляет собой цепь Маркова, для которой следующее положение зависит только от текущего местоположения (первое слагаемое в уравнении) и вероятности перехода (второе слагаемое).

Известно несколько модификаций алгоритма поиска кукушки. В каноническом алгоритме кукушка в своих полётах Леви никак не учитывает информацию о лучших найденных решениях. В модифицированном алгоритме поиска кукушки компоненты вектора V вычисляются по формуле вида [20]:

$$V = U_1(0; 1)(X^{best} - X_k), k \in [1: |S|],$$

где $X_k \neq X^{best}$ – случайно выбранное гнездо.

Так определенный вектор V обеспечивает большую вероятность полёта кукушки к гнёздам, имеющим высокую приспособленность. Вместе с тем, в каноническом алгоритме вероятность ε_a и параметры полёта Леви являются фиксированными константами. В целях диверсификации поиска на ранних итерациях целесообразно использовать большие значения величин ε_a , v . На завершающих итерациях для повышения точности локализации экстремума (интенсификации поиска) разумны меньшие значения указанных величин.

Улучшенный алгоритм поиска кукушки использует динамические значения этих параметров [20]:

$$\varepsilon_a(t) = \varepsilon_a^{max} - \frac{t}{\hat{t}}(\varepsilon_a^{max} - \varepsilon_a^{min});$$

$$v(t) = v^{max} \exp(d^t), d = \frac{1}{\hat{t}} \left(\frac{v^{min}}{v^{max}} \right).$$

Здесь ε_a^{min} , ε_a^{max} , v^{min} , v^{max} – заданные константы.

Некоторые из новых решений должны быть порождены полетами Леви вокруг текущего лучшего решения, что ускорит сходимость алгоритма. Значительная часть новых решений должна быть сформирована в случайно выбранных точках пространства поиска, т.е. достаточно далеко от текущих лучших решений. Это позволит диверсифицировать поиск, повысить вероятность преждевременной сходимости.

4.3. Алгоритм летучих мышей

Алгоритм летучих мышей был предложен Янгом (X.-Sh. Yang) в 2010 г. Большинство видов летучих мышей обладает удивительно совершенными средствами эхолокации, которая используется ими для обнаружения добычи и препятствий, а также для обеспечения возможности разместиться в темноте на насесте. Параметры лоцирующего звукового импульса мышей различных видов меняются в широких пределах, отражая их различные охотничьи стратегии. Большинство мышей используют короткие частотно-модулированные в пределах примерно одной октавы (музыкальный интервал) сигналы. В тоже время некоторые виды не используют частотную модуляцию своих звуковых импульсов [23].

Алгоритм предполагает следующую модель поведения летучих мышей [24]:

1. с помощью эхолокации все мыши могут измерить расстояние до добычи и препятствий, а также различать их;

2. мыши движутся случайным образом. Текущее положение и скорость мыши равны X_i, V_i соответственно. Для поиска добычи мыши генерируют сигналы, имеющие частоту ω_i , и громкость a_i . В процессе поиска мыши могут менять частоту этих сигналов, а также частоту повторения излучаемых импульсов (*rate of pulse*) $r \in [0; 1]$;

3. частота сигналов может изменяться в диапазоне $[\omega^{min}, \omega^{max}]$, $\omega^{max} > \omega^{min} \geq 0$, а громкость сигналов – в пределах $[0; 1]$. Положим, что речь идет о задаче глобальной безусловной минимизации (рис.4.2).

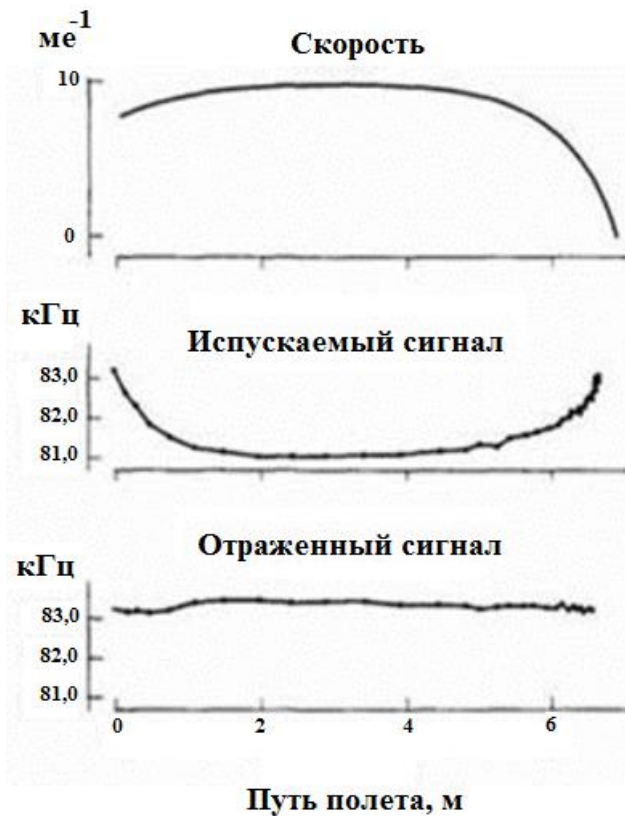


Рис. 4.2. Характеристики полета летучей мыши

Схема алгоритма включает следующие основные шаги [23]:

Шаг 1. Инициализируется популяция агентов s_i , $i \in [1: |S|]$. Определяем глобально лучшего агента s^{best} и соответствующее ему решение X^{best} ;

Шаг 2. Выполняется перемещение всех агентов на один шаг в соответствии с используемой миграционной процедурой;

Шаг 3. Для каждого из агентов s_i , $i \in [1: |S|]$, выполняем следующие действия:

– генерируем случайное число $u = U_1(0; 1)$;

- если $u > r_i$, то находим лучшее решение X_i^{best} данного агента;
- в окрестности решения X_i^{best} реализуем процедуру локального поиска.

Принимаем найденное решение в качестве текущего положения агента s_i ;

Шаг 4. В окрестности текущего решения случайным образом генерируем новое решение;

Шаг 5. Генерируем новое случайное число $u = U_1(0; 1)$;

Шаг 6. Если $u < a_i$ и $\varphi(X_i) < \varphi(X^{best})$, то принимаем решение X_i в качестве нового текущего положения агента s_i , увеличиваем значение параметра r_i и уменьшаем значение параметра a_i ;

Шаг 7. Находим новое глобально лучшее решение X^{best} ;

Шаг 8. Если условие окончания итераций не выполнено, то возвращаемся к шагу 2

На этапе инициализации алгоритма начальные значения частот ω_i^0 , громкостей a_i^0 , и частот повторения импульсов r_i^0 , $i \in [1: |S|]$, полагаем равномерно распределенными в соответствующих интервалах $[\omega^{min}, \omega^{max}]$, $[a^{min}, a^{max}]$, $[0; 1]$.

Миграцию агента s_i , $i \in [1: |S|]$, осуществляем по формулам:

$$\begin{aligned} X'_i &= X_i + V'_i, \\ V'_i &= Vi + \omega'_i(X_i - X^{best}), \\ \omega'_i &= \omega^{min} + (\omega^{max} - \omega^{min})U_1(0; 1). \end{aligned}$$

Другими словами, на данной итерации агент перемещается в направлении, определяемом суммой вектора перемещения на предыдущей итерации (слагаемое Vi) и случайным образом возмущённого вектора направления на лучшего агента ($X_i - X^{best}$). Заметим, что рассмотренная процедура миграции алгоритма летучих мышей имеет много общего с аналогичной процедурой алгоритма роя частиц.

Случайный локальный поиск выполняем по следующей схеме:

1) случайным образом варьируем текущее положение агента в соответствии с формулой:

$$X'_i = X_i + \bar{a}U_{|X|}(-1; 1), i \in [1: |S|],$$

где \bar{a} – текущее среднее значение громкостей всех агентов популяции;

$$\bar{a} = \frac{1}{|S|} \sum_{i=1}^{|S|} a_i$$

2) вычисляем значение фитнес-функции в новой точке $\varphi(X'_i) = \varphi'_i$. Если $\varphi'_i > \varphi_i$, то завершаем процедуру локального поиска, в противном случае фиксированное число раз возвращаемся к шагу 1.

Эволюция параметров a_i, r_i осуществляется по правилам:

$$a'_i = b_a a_i, r'_i = r_i^0 (1 - \exp(-b_r t)), i \in [1: |S|],$$

где $b_a \in (0; 1), b_r > 0$ – заданные константы (свободные параметры алгоритма), рекомендуемые значения которых равны 0,9. Другими словами, с ростом числа итераций громкость импульсов, излучаемых каждой мышью, линейно уменьшаем (добыча уже найдена), а частоту повторения импульсов в тех же условиях уменьшаем по экспоненциальному закону, так что имеют место предельные соотношения

$$a_i^t \xrightarrow{i \rightarrow \infty} 0, r_i^t \xrightarrow{i \rightarrow \infty} r_i^0, i \in [1: |S|]$$

4.4. Алгоритм обезьяньего поиска

Известны два существенно различных алгоритма, вдохновленных некоторыми аспектами поведения обезьян – алгоритм обезьяньего поиска и обезьяний алгоритм.

Алгоритм обезьяньего поиска предложили Мучерино (*A. Mucherino*) и Шереф (*O. Seref*) в 2008 году. Алгоритм вдохновлен поведением обезьяны, лазающих по дереву в поисках пищи. Обезьяне ставится соответствие агент, который строит деревья решений для поиска экстремума в задаче глобальной максимизации [25].

В алгоритме максимальное количество пищи представляет собой желаемое решение, а ветви дерева представляют собой варианты выбора между соседями допустимыми решениями в рассматриваемой задаче оптимизации. Этот выбор может быть полностью случайным, так и основанным на известных алгоритмах решения задачи глобальной оптимизации. Алгоритм использует бинарные деревья поиска, т.е. от каждой данной ветки (кроме ветвей, образующих вершину дерева) отходят две другие ветви с решениями, располагающимися на их концах.

Канонический алгоритм обезьяньего поиска не является популяционным (роевым) поскольку предлагает поиск с помощью только одной обезьяны. Известны «многообезьяньи» модификации канонического алгоритма, что и позволяет включить данный алгоритм в рассмотрение.

Если в текущий момент обезьяна находится на конце некоторой ветки, то далее она с равной вероятностью перемещается по левой и правой исходящим ветвям. В точке пространства поиска, соответствующей концу ветки, на которой находится обезьяна, вычисляем значение фитнес-функции. Если это решение лучше найденного ранее лучшего решения, то оно запоминается, и по рассмотренной схеме обезьяна продолжает движение вверх. Движение останавли-

вается при достижении обезьяной вершины дерева, определяемой максимально допустимой его высотой l_{max} . Все посещенные ветви запоминаются.

Если не все пути на дереве исследованы, то всякий раз, после достижения обезьяной вершины дерева, она спускается до текущей лучшей точки и снова начинает движение вверх, возможно, проходя некоторые из уже пройденных ветвей [25].

Принципиальным в алгоритме является способ генерации ветвей дерева поиска, т.е. генерации решений, соответствующих с текущим положением обезьяны. Эти решения получают путем локального возмущения текущего решения, для чего могут быть использованы миграционные операторы любых популяционных алгоритмов. Например, если X, X^{best} – текущее и лучшее положения обезьяны, то два следующих ее возможных положения X'_1, X'_2 могут быть найдены путем скрещивания решений X, X^{best} . Известны модификации алгоритма обезьяньего поиска, использующие для генерации ветвей алгоритмы колонии муравьев, поиска гармонии, и т.д.

Для уменьшения вероятности преждевременной стагнации поиска может быть ограничено максимальное число путей, подлежащих исследованию на данном дереве. Если число достигнуто, то лучшее полученное решение дерева запоминается. Затем из семени – случайной точки пространства – «выращивается» новое дерево поиска.

После исследования по рассмотренной схеме заданного числа деревьев в качестве семян новых деревьев могут быть использованы случайные решения из списка лучших решений. Поиск останавливаем, когда исследовано все предопределенное число деревьев.

Обезьяний алгоритм предложили Жао (*R. Zhao*) и Танг (*W. Tang*) в 2007 году. Алгоритм моделирует поведение обезьян в процессе их лазания по горам в целях поиска пищи. Полагают, что обезьяны исходят из того, что чем выше гора, тем больше пищи на ее вершине. Местность, которую обследуют обезьяны, представляет собой ландшафт фитнес-функции, так что решению соответствует самая высокая гора (рассматриваем задачу глобальной максимизации).

Из своего текущего положения каждая из обезьян движется вверх до тех пор, пока не достигнет вершины горы. Затем, обезьяна делает серию локальных прыжков в случайном направлении в надежде найти более высокую гору, и движение вверх повторяется [26].

После выполнения некоторого числа подъемов и локальных прыжков, обезьяна полагает, что в достаточной степени исследовала ландшафт в окрест-

ности своего начального положения. Для того, чтобы обследовать новую область пространства поиска, обезьяна выполняет длинный глобальный прыжок.

Указанные действия повторяются заданное число раз. Решением задачи объявляется самая высокая из вершин, найденных данной популяцией обезьян.

Процесс движения вверх представляет собой процесс локального поиска и может быть реализован многими способами. В оригинальном алгоритме авторы предлагают использовать алгоритм на основе процедуры стохастической аппроксимации, при котором каждое следующее значение оценки получается в виде основанной лишь на новом наблюдении поправки к уже построенной оценке.

Алгоритм локальных прыжков для агента s_i , $i \in [1: |S|]$, имеет следующий вид [26]:

Шаг 1. Исходя из текущего положения агента X_i новое положение X'_i по формуле

$$x'_{i,j} = U_1((x_{i,j} - b); (x_{i,j} + b)), i \in [1: |S|], j \in [1: |X|],$$

т.е. полагаем, что по каждой из координат новое положение агента представляет собой новую величину, равномерно распределенную на интервале $[(x_{i,j} - b); (x_{i,j} + b)]$. Здесь $b > 0$ – максимально возможная длина прыжка вдоль соответствующего координатного направления (свободный параметр алгоритма).

Шаг 2. Если точка X'_i является допустимой (принадлежит области допустимых значений D) и $\varphi(X'_i) \geq \varphi(X_i)$ то полагаем $X_i = X'_i$ и завершаем для данного агента локальные прыжки, в противном случае возвращаемся к шагу 1.

Глобальные прыжки агенты выполняют из своего текущего положения в направлении центра текущего центра тяжести их координат по следующей схеме:

Шаг 3. Генерируем случайное число v , равномерно распределенное на интервале $[v_{min}, v_{max}]$ и имеющее смысл глобального прыжка:

$$v = U_1(v_{min}, v_{max})$$

Здесь v_{min}, v_{max} – нижняя и верхняя границы этой величины, которые в общем случае имеют разные знаки, так что величина v может быть как положительной так и отрицательной. Для диверсификации поиска интервал $[v_{min}, v_{max}]$ следует расширять, а для интенсификации – сужать.

Шаг 4. Находим новое возможное положение X'_i агента s_i по формуле:

$$x'_{i,j} = x_{i,j} + v(x_{i,j}^c - x_{i,j}), i \in [1: |S|], j \in [1: |X|],$$

где $x_{i,j}^c = \frac{1}{|S|} \sum_{i=1}^{|S|} x_{i,j}$ – текущее положение центра тяжести агентов популяции по j -му координатному направлению.

Шаг 5. Если положение X'_i является допустимым, то полагаем $X_i = X'_i$ и завершаем для данного агента глобальные прыжки, в противном случае возвращаемся к шагу 1.

Отметим, что если величина v принимает положительное значение, то агент совершает прыжок в сторону центра тяжести, а если отрицательное значение – в противоположную сторону. Поскольку алгоритм не гарантирует, что лучшее решение будет получено на последней итерации, каждый раз необходимо сохранять в памяти ЭВМ текущее лучшее значение.

4.5. Алгоритм прыгающих лягушек

Тасующий алгоритм прыгающих лягушек предложил Юсуф (M.Eusuff) с соавторами в 2003 году. По сути, алгоритм представляет собой гибридизацию меметического алгоритма и алгоритма роя частиц. Понятие меметики определено, как подход к эволюционным моделям передачи информации, который основывается на концепции мемов, являющимися аналогами генов, – как единицы культурной информации, распространяемой между людьми посредством имитации, научения и др. Понятие мема и основы меметики разработаны, Докином (*C.R. Dawkins*) в 1976 году. Тасующий алгоритм прыгающих лягушек вдохновлен поведением группы лягушек в процессе поиска пищи [27].

Рассмотрим задачу глобальной условной максимизации. Алгоритм включает в себя следующие шаги [28]:

Шаг 1. Инициализируется популяция $S = (s_i, i \in (1: |S|))$;

Шаг 2. Оценивается пригодность агентов популяции $\varphi(X_i) = \varphi_i, i \in (1: |S|)$ и определяем на этой основе глобального лучшего агента s^{best} ;

Шаг 3. Агенты разделяются на $|S^p|$ наборов $S_j^p = s_j, p \in (1: n), j \in (1: |S^p|)$, названных авторами алгоритма мемеплексами, каждый из которых содержит по $n = \frac{|S|}{|S^p|}$ агентов, так что $\bigcup_{i=1}^{|S^p|} S_j^p = S$. Разделение выполняется по следующему правилу соседства: агента s_1 относится к мемеплексу S_1^p , агента s_2 относится к мемеплексу S_2^p и т.д. до мемеплекса S_n^p , к которому относится агент s_n ; агента s_{n+1} – к мемеплексу S_1^p ; агента s_{n+2} – к мемеплексу S_2^p и т.д. до последнего агента $s_{|S|}$, который должен относиться к мемеплексу S_n^p . Полагаем, что величины $|S|$ и $|S^p|$ кратны.

Шаг 4. Для каждого из мемеплексов выполняется процедура меметической эволюции, которая состоит в следующем:

а) в каждом из мемеплексов S_j^p находят лучший s_j^{best} и худший s_j^{worst} агенты;

b) пытаемся улучшить положение худшего агента путем случайного перемещения его в направлении к или от лучшего агента по формуле:

$$X_{j*}^{worst}(t+1) = X_{j*}^{worst} U_1(-0,5; 0,5) v_{max} \frac{X_{j*}^{best} - X_{j*}^{worst}}{\|X_{j*}^{best} - X_{j*}^{worst}\|_E}, j \in (1: |S^p|),$$

где v_{max} – максимально допустимое значение шага перемещения. Если $\varphi(X_{j*}^{worst}(t+1)) > \varphi(X_{j*}^{worst})$, то фиксируется удачное перемещение;

с) если предыдущая операция на привела к успеху, то по той же схеме пытаемся улучшить положение агента s_{j*}^{worst} путем перемещения его в направлении глобального лучшего агента s^{best} ;

d) если и последняя операция не привела к улучшению позиции агента s_i , то вместо этого агента случайным образом создаем в области поиска нового агента.

Шаг 5. Повторяется процедура меметической эволюции заданное число раз, реализуя тем самым локальный поиск в пределах каждого из мемеплексов.

Шаг 6. Выполняется процедура тасования агентов, которая заключается в объединении агентов всех мемеплексов в одну группу и случайном изменении их номеров. В результате при повторном выполнении шага 3 будут получены новые мемеплексы $S_j^p, j \in (1: |S^p|)$.

Шаг 7. Если условие завершения итераций не выполнено, то возвращаемся к шагу 2.

Таким образом, основой тасующего алгоритма прыгающих лягушек является комбинирование локального поиска в пределах каждого из мемеплексов и глобального поиска путем обмена информацией о положениях лучших агентов этих мемеплексов и определения на этой основе глобального лучшего агента.

Известны модификации алгоритма, в которых, например, на этапе локального поиска агент движется не точно в направлении лучшего агента соответствующего мемеплекса, а в некотором случайным образом возмущенном направлении. Известны как последовательные, так и параллельные гибридизации алгоритма со многими популяционными алгоритмами [27].

4.6. Алгоритм поиска косяком рыб

Алгоритм поиска косяком рыб предложили в 2008 году Фило (*B. Filho*) и Нето (*L. Neto*) [29].

Косяком рыб называют агрегацию рыб, которые передвигаются приблизительно с одной и той же скоростью, и ориентацией, поддерживая примерно постоянное расстояние между собой. Доказано, что всякого рода объединение

рыб играют важную роль в повышении эффективности поиска ими пищи, защиты от хищников, а также в уменьшении энергетических затрат.

В алгоритме рыбы плавают в аквариуме (области поиска) в поисках пищи (решение задачи оптимизации). Вес каждой рыбы формализует ее индивидуальный успех в поиске решения и играет роль ее памяти. Именно наличие веса у агентов популяции является главной парадигмой алгоритма поиска косяком рыб в сравнении, например, с парадигмой оптимизации роя частиц. Эта особенность алгоритма позволяет отказаться от необходимости отыскивать и фиксировать глобально лучшие решения, как это делается в алгоритме роя частиц.

Полагается, что решается задача глобальной условной максимизации в области D , и что всюду в этой области фитнес-функция принимает неотрицательные значения.

Операторы алгоритма поиска косяком рыб объединены в две группы [30]:

а) оператор кормления, формализующий успешность исследования агентами тех или иных областей аквариума;

в) операторы плавания, реализующие алгоритмы миграции агентов.

Оператор кормления. Обозначим $\omega_i, i \in [1: |S|]$, текущий вес агента s_i . В алгоритме поиска косяка рыб принято, что вес агента пропорционален нормализованной разности значений фитнес-функции на следующей и текущей итерациях, т.е.

$$\omega'_i = \omega_i + \frac{\varphi(X'_i) - \varphi(X_i)}{\max(\varphi(X'_i), \varphi(X_i))}, i \in [1: |S|]$$

Алгоритм ограничивает максимально допустимый вес агентов величиной $\omega_{max} > 0$, так что во всех случаях вес агента s_i удовлетворяет условию

$$\omega_i \in [1: \omega_{max}], i \in [1: |S|]$$

Здесь ω_{max} – свободный параметр алгоритма.

При инициализации популяции всем агентам в качестве веса присваивается значение $\frac{\omega_{max}}{2}$.

Операторы плавания. В алгоритме поиска косяком рыб различают три вида плавания – индивидуальное, инстинктивно-коллективное, коллективно-волевое. Положим, что эти виды плавания выполняются в интервалах $(t, \tau], (\tau, \theta], (\theta, t']$ основного интервала $(t, t']$ соответственно. Здесь $t < \tau < \theta < t', t' = t + 1$.

а) Индивидуальное плавание. Направление перемещения агента в этом случае равновероятно случайно. Если это перемещение выводит агента за пределы области допустимых значений D , то перемещение не выполняем. Аналогично, перемещение агента s_i не проводится, если имеет место неравенство

$X_i^\tau < X_i^t$, т.е. в новой точке X_i^τ значение фитнес-функции не выше ее значения в предыдущей точке X_i^t , $i \in [1: |S|]$.

Компоненты шага перемещения V_i^{ind} полагаем случайными величинами равномерно распределенными на интервале $[0; v_{max}^{ind}]$:

$$V_i^{ind} = U_{|X|}(0; 1)v_{max}^{ind}, i \in [1: |S|]$$

Для обеспечения постепенного перехода от диверсификации поиска на начальных итерациях к его интенсификации на завершающих итерациях линейно уменьшаем величину v_{max}^{ind} (свободный параметр алгоритма) с ростом числа итераций.

Процесс индивидуального плавания может включать в себя не одну итерацию, а некоторое их фиксированное число. Таким образом, индивидуальное плавание агента можно интерпретировать как локальный поиск в окрестностях текущего положения агента [29].

в) Инстинктивно-коллективное плавание реализуется после завершения всеми $|S|$ агентами индивидуальных плаваний по формуле.

$$X_i^\theta = X_i^\tau + \frac{\sum_j V_j^{ind}(\tau)(\varphi(X_i^\tau) - \varphi(X_i^t))}{\sum_j (\varphi(X_i^\tau) - \varphi(X_i^t))}, i \in [1: |S|]$$

Второе слагаемое в формуле есть нечто иное, как общий для всех агентов шаг миграции, представляющий собой взвешенную сумму, индивидуальных перемещений агентов. Это означает, что в процессе инстинктивно-коллективного плавания на каждого из агентов оказывают влияние все остальные агенты популяции, и это влияние пропорционально индивидуальным успехам агентов [29].

с) Коллективно-волевое плавание выполняется вслед за инстинктивно-коллективным плаванием. Коллективно-волевое плавание заключается в смещении всех агентов в направлении текущего центра тяжести популяции, если суммарный вес косяка в результате индивидуального и инстинктивно-коллективного плавания увеличится, и в противоположном направлении – если вес уменьшится. Другими словами, в случае успешных в среднем указанных плаваний популяция стягивается к своему центру тяжести, т.е. повышается интенсивность поиска. В противном случае популяция расширяется от того же центра, повышая свои диверсификационные свойства [29].

Координаты центра тяжести X_c косяка после завершения всеми его агентами инстинктивно-коллективных плаваний определяем по формуле [30]:

$$X_c^\theta = \frac{\sum_i w_i^\theta X_i^\theta}{w_\Sigma^\theta}, i \in [1: |S|]$$

где $w_{\Sigma}^{\theta} = w_{\Sigma}(\theta) = \sum_{i=1}^{|S|} w_i^{\theta}$ – текущий суммарный вес популяции.

Коллективно-волевое плавание выполняется по правилу [30]:

$$X'_i = X_i^{\theta} \pm v^{vol}(X_c^{\theta} - X_c^{\theta}), i \in [1: |S|]$$

где знак плюс используется в случае $w_{\Sigma}^{\theta} > w_{\Sigma}^{\theta-1}$, а знак минус – в противном случае. Здесь $w_{\Sigma}^{\theta-1}$ – суммарный вес популяции после завершения ее агентами инстинктивно-коллективных плаваний на предыдущей итерации.

Переменная v^{vol} в формуле определяет размер шага перемещений агентов и является случайной величиной:

$$v^{vol} = v_{max}^{vol} U_1(0; 1)$$

Здесь $v_{max}^{vol} > 0$ – заданная максимально допустимая длина шага (свободный параметр алгоритма). Рекомендуется линейное уменьшение величины v_{max}^{vol} с ростом числа итераций.

Известен ряд модифицированных алгоритмов поиска косяком рыб. В качестве примера можно привести так называемый плотный алгоритм, использующий модификацию операторов классического алгоритма, а также новые операторы памяти и разбиения.

Оператор памяти строится на основе $(|S| \times 1)$ -векторов памяти $M_i, i \in [1: |S|]$, агентов популяции и формализует текущее и ряд предшествующих влияний на каждого данного агента других агентов популяции.

Оператор разбиения использует память агентов и предназначен для формирования на основе популяции $S = (s_i, i \in [1: |S|])$ ряда популяций.

Известны также высокоэффективные гибридные алгоритмы, построенные на основе алгоритм поиска косяком рыб. Примером такой гибридизации может служить волевой алгоритм роя частиц [29].

4.7. Сорняковый алгоритм

Алгоритм сорняковой оптимизации вдохновлен таким общераспространенным явлением, как колонизация сельскохозяйственных угодий сорняками. Алгоритм предложен в 2006 году Мехрабиан (A.R. Mehrabian) и Лукас (C. Lucas) [31].

Сорняк – это любое растение, растущее там, где оно не желательно. В общем случае любое растение может быть классифицировано как сорняк. Однако обычно термин используется по отношению к тем растениям, чьи экспансионистские свойства представляют серьезную угрозу для культивируемых растений.

Основным механизмом, определяющим динамику сообщества любых растений, является естественный отбор, из которого выделяют два крайних ти-

па: r -отбор и K -отбор. Реальные стратегии отбора лежат между этими предельными типами.

Можно сказать, что девизом r -отбора являются слова «живи быстро, размножайся быстро, умирай молодым». Данный тип отбора необходим для успеха в нестабильной, непредсказуемой окружающей среде. При r -отборе предпочтительны такие качества, как высокая плодовитость, маленький размер семян и приспособленность к рассеиванию их на большое расстояние.

K -отбор использует принцип «живи медленно, размножайся медленно, умирай в старости». Этот тип отбора для успеха в стабильной, предсказуемой среде, когда вероятно тяжелое соперничество за ограниченные ресурсы между конкурентно способными индивидуумами. Ситуация имеет место, если размер популяции в ареале обитания близок к максимуму, который он способен вместить. При K -отборе предпочтительны такие качества индивидов, как большой размер семян, длинная жизнь, небольшое потомство, за которым требуется интенсивный уход.

Алгоритм рассматривает задачу глобальной безусловной минимизации. Модель поведения сорняков при колонизации учитывает следующие базовые шаги этого процесса [31]:

Шаг 1. Распределение конечного числа семян по всей области поиска (инициализация популяции).

Шаг 2. Производство выросшими растениями семян в зависимости от приспособленности растений (воспроизводство).

Шаг 3. Размещение произведенных семян в случайном порядке по области поиска (пространственное распределение).

Шаг 4. Повторение шагов 2, 3 до тех пор, пока не достигнут заданный максимум числа растений.

Шаг 5. Отбор растений с более высокой приспособленностью, их воспроизводство и пространственное распределение.

Шаг 6. Повторение шага 5 до выполнения условия окончания процесса.

Воспроизводство. В оригинальном алгоритме сорняковой оптимизации число семян n_i^S , произведенных сорняком $s_i, i \in [1: |S|]$, линейно зависит от его текущей приспособленности $\varphi_i = \varphi(X_i)$ и определяется формулой [31]:

$$n_i^S = \frac{n_{max}^S - n_{min}^S}{\varphi^{best} - \varphi^{worst}} \varphi_i + \frac{\varphi^{best} n_{min}^S - \varphi^{worst} n_{max}^S}{\varphi^{best} - \varphi^{worst}},$$

n_{max}^S, n_{min}^S – заданные контакты, представляющие собой максимальное и минимальное число семян, которые могут быть произведены каждым из сорняков на текущей итерации.

Пространственное распределение. Произведенные сорняком $s_i, i \in [1: |S|]$, семена распределяем в окрестности родительского растения в соответствии с нормальным законом распределения:

$$X_{i,j} = X_i + N_{|X|}(0, \sigma), i \in [1: |S|], j \in [1: n_i^S]$$

Стандартное отклонение σ в последнем выражении зависит от текущего номера поколения t , уменьшаясь с ростом этого номера по формуле [1]:

$$\sigma = \sigma(t) = \left(\frac{\hat{t} - t}{\hat{t}} \right)^m (\sigma_b - \sigma_e) + \sigma_e,$$

где σ_b, σ_e – начальное и конечное значения стандартного отклонения σ , $m > 1$ – свободный параметр, определяющий характер функции $\sigma(t)$ (параметр модуляции).

Последняя формула обеспечивает уменьшение вероятности попадания семян вдали от родительского растения с ростом номера итераций, уменьшая тем самым диверсификационные и повышая интенсификационные свойства алгоритма. Можно сказать, что данная схема изменения стандартного отклонения σ реализует механизм перехода от r -отбора к K -отбору в процессе эволюции популяции.

Конкурентное исключение. Обозначим S_i^S популяцию сорняков, являющихся потомками растения s_i . Тогда до достижения всей популяцией сорняков своего максимального размера, равного $|S|$, новую популяцию S' формируется путем объединения текущей популяции S со всеми популяциями S_i^S [1]:

$$S' = \cup \left(\cup_{i=1}^{|S|} S_i^S \right)$$

Конкурентное исключение начинает функционировать после достижения популяцией размера $|S|$ и заключается в уничтожении сорняков с меньшей приспособленностью до достижения популяцией размера $|S|$. Таким образом, растения и их потомки оцениваются вместе, и тем, которые обладают лучшей приспособленностью, позволяют размножаться. Данный механизм позволяет растениям с меньшей приспособленностью воспроизводиться, они могут выжить.

4.8. Бактериальная оптимизация

К настоящему времени разработано несколько алгоритмов оптимизации, объединяемых общим названием бактериальная оптимизация). Прямо считать, что метод бактериальной оптимизации предложен Пассино (*K. M. Passino*) в 2002 году [32].

Подвижные бактерии, такие как кишечная палочка или сальмонелла, продвигают себя с помощью вращающихся жгутиков. Чтобы двигаться вперед, все жгутики вращаются в одном направлении. При этом бактерия совершает дви-

жение, которое называют плавание. При вращении жгутиков в разные стороны бактерия разворачивается – совершает движение кувырок.

Поведение бактерий обусловлено механизмом, который называется бактериальным хемотаксисом и представляет собой двигательную реакцию микроорганизмов на химический раздражитель. Данный механизм позволяет бактерии двигаться по направлениям к аттрактантам (чаще всего, питательным веществам) и от репеллентов (потенциально вредных для бактерии веществ).

Если выбранное бактерией направление движения соответствует увеличению концентрации аттрактанта (снижению концентрации репеллента), то время до следующего кувырка увеличивается. В силу малого размера бактерии сильное влияние на её перемещения оказывает броуновское движение. В результате бактерия только в среднем движется в направлениях к полезным веществам и от вредных веществ.

В контексте задачи поисковой оптимизации бактериальный хемотаксис можно интерпретировать как механизм оптимизации использования бактерией известных пищевых ресурсов и поиска новых, потенциально более ценных областей.

Канонический алгоритм бактериальной оптимизации. Рассмотрим задачу многомерной глобальной безусловной максимизации [1].

Канонический алгоритм бактериальной оптимизации основан на использовании трёх следующих основных механизмов: хемотаксис, репродукция, ликвидация и рассеивание.

Пусть $X_{i,r,l}$ – $(|X| \times 1)$ -вектор текущего положения бактерии $s_i \in S$ на итерации t (на t -м шаге хемотаксиса), r -м шаге репродукции и l -м шаге ликвидации и рассеивания. Здесь $i \in [1:|S|]$, $t \in [1:\hat{t}]$, $r \in [1:\hat{t}^r]$, где $|S|$ – чётное число агентов в колонии бактерий S ; \hat{t} , \hat{t}^r , \hat{t}^l – общие числа итераций (шагов хемотаксиса), шагов репродукции, а также шагов ликвидации и рассеивания. Соответствующее значение фитнес-функции обозначим $\varphi_{i,r,l}$.

Хемотаксис. Процедура хемотаксиса реализует в алгоритме бактериальной оптимизации локальную оптимизацию. Следующее положение $X'_{i,r,l}$ бактерии s_i определяет формула [1]:

$$X'_{i,r,l} = X_{i,r,l} + \lambda_i \frac{V_i}{\|V_i\|_E}$$

где V_i – текущий направляющий $(|X| \times 1)$ -вектор шага хемотаксиса бактерии s_i ; λ_i – текущее значение этого шага. При плавании бактерии на следующей итерации вектор V_i остается неизменным, т. е. имеет место равенство $V'_i = V_i$. При кувырке бактерии вектор V'_i представляет собой случайный век-

тор, компоненты которого имеют значения в интервале $[-1; 1]$. Другими словами, при кувырке имеет место равенство $V_i' = U_{|X|}(-1; 1)$. Плавание каждой из бактерий продолжается до тех пор, пока значения фитнес-функции увеличиваются.

Значение шага хемотаксиса в формуле может меняться в процессе поиска, уменьшаясь по некоторому закону с ростом числа итераций t .

Репродукция. Механизм репродукции имеет своей целью ускорение сходимости алгоритма (за счёт сужения области поиска). Назовём текущим состоянием здоровья h_i бактерии s_i сумму значений фитнес-функции во всех точках её траектории от первой до текущей итерации t [1]:

$$h_i = \sum_{\tau=1}^t \varphi_{i,r,l}(\tau).$$

Вычислим значения $h_i, i \in [1: |S|]$, отсортируем все бактерии в порядке убывания состояний их здоровья и представим результат сортировки в виде линейного списка. Механизм репродукции состоит в том, что на $(r + 1)$ -м шаге репродукции вторая половина агентов (наиболее слабых) исключается из указанного списка (погибает), а каждый из агентов первой (выжившей) половины списка расщепляется на два одинаковых агента с одинаковыми координатами, равными координатам расщеплённого агента.

Пусть, например, $s_j, j \in [1: |S|]$, – один из выживших агентов, положение которого определяет вектор $X_{i,r,l}$. После репродукции этого агента получаем агентов $s_j, s_k, k = |S|/2 + j$ положения которых равны $X'_{j,(r+1),l} = X_{i,r,l} X'_{k,(r+1),l} = X_{j,r,l}$.

В результате выполнения процедуры репродукции результирующее число бактерий в популяции остаётся неизменным и равным $|S|$.

Ликвидация и рассеивание. Рассмотренных процедур хемотаксиса и репродукции в общем случае недостаточно для отыскания глобального максимума многоэкстремальной целевой функции, поскольку эти процедуры не позволяют бактериям покидать найденные ими локальные максимумы этой функции. Процедура ликвидации и рассеивания призвана преодолеть этот недостаток.

Механизм ликвидации и рассеивания включается после выполнения определённого числа процедур репродукции и состоит в следующем. С заданной вероятностью ε_e случайным образом выбираем n бактерий $s_{i_1}, s_{i_2}, \dots, s_{i_n}$ и уничтожаем их. Вместо каждой из уничтоженных бактерий в случайно выбранной точке пространства поиска создаём нового агента с тем же номером. В ре-

результате выполнения операции ликвидации и рассеивания число бактерий в колонии также остаётся постоянным и равным $|S|$.

Более строго процедуру ликвидации и рассеивания определяет следующая последовательность шагов [32]:

Шаг 1. полагаем $j = 1$;

Шаг 2. генерируем натуральное случайное число $i_j = U_l(1: |S|)$ и вещественное случайное число $u_j = U_l(0; 1)$;

Шаг 3. если $u_j > \varepsilon_e$, то новые координаты бактерии s_j определяем по формуле:

$$x_{i,j,r,l,k} = U_l(x_k^-; x_k^+), k \in [1: |X|]$$

где, x_k^-, x_k^+ – константы, определяющие диапазон возможных начальных координат бактерий;

Шаг 4. полагаем $j = j + 1$ и продолжаем итерационный процесс до тех пор, пока не будет уничтожено n бактерий.

Определение векторов начальных положений бактерий $X_{i,0,0}(0), i \in [1: |S|]$ выполняется по формуле, аналогичной формуле на шаге 3.

Кооперативная бактериальная оптимизация. Предполагает разделение процедуры оптимизации колонией бактерий на несколько этапов, каждый из которых занимает некоторую часть общего числа поколений и характеризуется различными значениями параметра λ . Известны два варианта кооперативного бактериального алгоритма оптимизации (*Cooperative Bacterial Foraging Optimization, CBFO*) – алгоритмы *CBFO-S* и *CBFO-H*. Рассмотрим вначале первый из указанных алгоритмов [1].

Последовательный алгоритм бактериальной оптимизации (CBFO-S) представляет собой пример гибридизации по схеме препроцессор/постпроцессор. Алгоритм использует несколько алгоритмов бактериальной оптимизации, отличающихся значениями параметра λ и выполняющихся последовательно друг за другом. Выход предыдущего алгоритма бактериальной оптимизации (лучшие позиции, найденные каждой из бактерий) поступают на вход следующего алгоритма бактериальной оптимизации. На начальном этапе используется алгоритм с большим значением параметра λ , обеспечивающим поиск по всему пространству поиска. В процессе поиска каждая из бактерий сохраняет в своей памяти координаты посещённых точек, а также соответствующие значения фитнес-функции. Позиция с наибольшим значением фитнес-функции объявляется многообещающим участком [1].

При переходе на следующий этап (к выполнению следующего алгоритма бактериальной оптимизации) текущее значение параметра λ по некоторому

правилу уменьшается и начинается поиск из точек, принадлежащих найденным на предыдущем этапе многообещающим позициям. Поиск с данным значением λ продолжается до выполнения условия перехода к следующему этапу. Затем процесс повторяется ещё меньшим значением параметра λ и так далее до выполнения условия окончания итераций. Число этапов алгоритма $n_p < \hat{t}$ является свободным параметром. Числа шагов хемотаксиса в пределах каждого из этапов обычно полагают одинаковыми. Новое значение шага λ определяют, например, по правилу:

$$\lambda' = v\lambda$$

где $v \in (0; 1)$ – свободный коэффициент уменьшения шага алгоритма.

Развитием алгоритма *CBFO-S* можно считать *адаптивный алгоритм бактериальной оптимизации (Adaptive Bacterial Foraging Optimization. ABFO)*, в котором шаг λ меняется, вообще говоря, на каждой итерации по правилу [32]:

$$\lambda' = \lambda(t + 1) = \sum_{\tau=0}^m b_{\tau} \lambda(t - \tau)$$

где m – число предыдущих учитываемых шагов, b_{τ} – весовые коэффициенты, назначаемые лицом, принимающим решения.

Гибридный алгоритм бактериальной оптимизации (CBFO-H) является собой пример низкоуровневой гибридизации вложением. Алгоритм является многопопуляционным и выполняется в два этапа. На первом этапе используем алгоритм бактериальной оптимизации с большим значением параметра λ , который после заданного числа итераций обнаруживает аналогично алгоритму *BFO-S* многообещающие участки и передаёт их на следующий этап.

Основной особенностью второго этапа алгоритма *BFO-H* является разложение пространства поиска $R^{|X|}$ на $\frac{|X|}{2}$ подпространств, каждое из которых имеет размерность равную двум. Поиск в каждом из указанных двумерных подпространств выполняет своя колония бактерий, совокупность которых образует мегаколонию $S = \{S_i, i \in [1: |X|/2]\}$, где $|X|/2 = |S|$ – число колоний в мегаколонии.

Алгоритм, использующий эффект роения бактерий. Эффект роения бактерий наблюдается для нескольких разновидностей бактерий, включая бактерию кишечной палочки и бактерию сальмонеллы, в полутвердой питательной среде. Эффект обусловлен локальным уменьшением концентрации аттрактанта вследствие его потребления бактериями. В результате возникает градиент этой концентрации и в хаотическом перемещении бактерий появляется составляющая, направленная по градиенту. Кроме того, бактерии продолжают размно-

жаться. Можно считать, что данный эффект объясняется наличием неявного канала связи между бактериями. В рассматриваемом алгоритме этот канал формализуют следующим образом.

Введем в рассмотрение величины d_{att} , w_{att} определяющие размеры области наличия аттрактанта, а также аналогичные величины d_{rep} , w_{rep} , характеризующие размеры области репеллентов (имеются в виду двумерная модель популяции бактерий). Указанную выше связь между бактериями задаем функцией [1].

$$\tilde{\varphi}(X) = \sum_{i=1}^{|S|} [-d_{att} \exp(-w_{att} \|X - X_i\|_E^2) + d_{rep} \exp(-w_{rep} \|X - X_i\|_E^2)]$$

Легко видеть, что алгебраическое значение первого слагаемого в формуле возрастает с увеличением евклидова расстояния от точки X до точки X_i . Наоборот, второе слагаемое в этой формуле убывает с ростом указанного расстояния. Таким образом, первая сумма в формуле формализует «притяжение» бактерий областью, в которой имеются запасы аттрактанта (а значит, и сосредоточены агенты популяции). Аналогично вторая сумма в этой формуле формализует «отталкивание» бактерий этой областью. Отметим, что значение функции $\tilde{\varphi}(X)$ не зависит от значения целевой функции в точке X .

В качестве фитнес-функции используется функция [1]

$$\varphi(X) := \varphi(X) + \tilde{\varphi}(X)$$

Варьируя значения параметров d_{att} , w_{att} , d_{rep} , w_{rep} можно управлять поведением колонии бактерий. Так, уменьшение параметра d_{att} , и увеличение параметра w_{rep} , а также увеличение d_{rep} , и уменьшение w_{rep} , увеличивает модифицированную фитнес-функцию и диверсифицирует поиск. Противоположные изменения указанных величин локализуют популяцию в небольших областях, обеспечивая высокую точность поиска

Гибридные бактериальные алгоритмы. Известно большое число алгоритмов оптимизации, построенных на основе гибридизации бактериального алгоритма с другими популяционными алгоритмами. Рассмотрим в качестве примеров низкоуровневую гибридизацию бактериального алгоритма с генетическим алгоритмом (алгоритм *BFO-GA*) и алгоритмом роя частиц (алгоритм *BFO-PSO*). Гибридный алгоритм *BFO-GA* представляет собой расширенный генетическими операторами отбора, скрещивания и мутации алгоритм, использующий роение бактерий. В качестве оператора отбора алгоритм *BFO-GA* использует алгоритм на основе метода рулетки. Скрещивание реализовано с помощью опе-

ратора арифметического кроссовера, а мутация – с помощью неравномерного мутатора Михалевича [1].

Указанные модифицирующие операторы выполняются после завершения процедур хемотаксиса и репродукции бактериального алгоритма перед процедурами ликвидации и рассеивания этого алгоритма. Основной целью гибридизации в алгоритме *BFO-PSO* было расширение возможностей агентов бактериального канонического алгоритма средствами обмена информацией между ними, позаимствованными в алгоритме роя частиц. Таким образом, алгоритм *BFO-PSO* моделирует процессы хемотаксиса, репродукции, ликвидации и рассеивания колонии бактерий, а также процесс роевания агентов.

4.9. Алгоритм гравитационного поиска

Алгоритм гравитационного поиска предложил Рашеди (*E.Rashedi*) с соавторами в 2009 году. Основу гравитационного поиска составляют законы гравитации и взаимодействия масс. Данный алгоритм похож на методы роя частиц, так как базируется на развитии многоагентной системы [33].

Алгоритм гравитационного поиска оперирует двумя законами:

1) законом тяготения (рис. 4.3): каждая частица притягивает другие и сила притяжения между двумя частицами прямо пропорциональна произведению их масс и обратно пропорциональна расстоянию между ними (следует обратить внимание на то, что в отличие от всемирного закона тяготения используется не квадрат расстояния. Рашеди объясняет это тем, что во всех тестах это дает лучшие результаты);

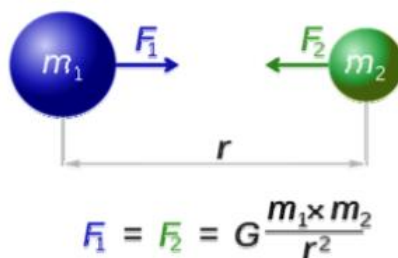


Рис. 4.3. Закон тяготения.

2) законом движения: текущая скорость любой частицы равна сумме части скорости в предыдущий момент времени и изменению скорости, которое равно силе, с которой воздействует система на частицу, делённой на инерциальную массу частицы.

Имея в арсенале эти два закона, метод работает последующему плану [33]:

Шаг.1. генерация системы случайным образом;

Шаг.2. определение приспособленности каждой частицы;

Шаг.3. обновление значений гравитационной постоянной, лучшей и худшей частиц, а также масс;

Шаг.4. подсчет результирующей силы в различных направлениях;

Шаг.5. подсчет ускорений и скоростей;

Шаг.6. обновление позиций частиц;

Шаг.7. повторений шагов 2 – 6 до выполнения критерия окончания (либо превышение максимального количества итераций, либо слишком малое изменение позиций, либо любой другой осмысленный критерий).

Сила, действующая в момент времени t на i -ю частицу со стороны j -й, рассчитывается по формуле [33]:

$$F_{ij}(t) = G(t) \frac{M_{pi}(t) \times M_{aj}(t)}{\|p_i, p_j\| + \varepsilon} (p_j(t) - p_i(t)),$$

где M_{aj} – активная гравитационная масса j -й частицы; M_{pi} – пассивная гравитационная i -й частицы; $G(t)$ – гравитационная постоянная в соответствующий момент времени; ε – малая константа; $\|.,.\|$ – евклидово расстояние между частицами.

Предположим, есть некоторая функция, которую необходимо минимизировать: $f(x): R^n \rightarrow R$. Кроме этого, есть область R , в которой генерируются начальные позиции частиц. В соответствии с планом работы гравитационного поиска, начинается все с генерации системы частиц [33]:

$$S = \{p_i = \{p_i^1, p_i^2, \dots, p_i^n\} \in X\}_{i=1}^N,$$

где N — максимальное количество частиц в системе.

Чтобы алгоритм был не детерминированным, а стохастическим, в формулу расчета результирующей силы $F_i(t)$ добавляются случайные величины ε_j (равномерно распределенные от нуля до единицы). Тогда результирующая сила равна:

$$F_i(t) = \sum_{j=1, j \neq i}^N \varepsilon_j F_{ij}(t)$$

Посчитаем ускорения и скорости [33]:

$$v_i(t+1) = (\varepsilon_1, \dots, \varepsilon_n)^T * v_i(t) + \frac{F_i(t)}{M_{ii}(t)},$$

где $*$ — операция покомпонентного умножения векторов; ε_i – случайная величина, равномерно распределенная от нуля до единицы; $M_{ii}(t)$ – инертная масса i -й частицы.

Далее необходим пересчет положения частиц:

$$p_i(t+) = p_i(t) + v_i(t+1).$$

Остаётся два вопроса: как изменяется гравитационная постоянная и как рассчитывать массы частиц. Значение гравитационной постоянной должно определяться монотонно убывающей функцией, зависящей от начального значений постоянной G_0 и момента времени t , т.е. $G(t) = G(G_0, t): R^+ \rightarrow R^+$.

Далее можно приступить к пересчёту масс. В простейшем случае все три массы (пассивная, активная и инерциальная) приравняются:

$$M_{ai} = M_{pi} = M_{ii} = M_i$$

Тогда значение масс можно пересчитать по формуле [33]:

$$M_i(t) = \frac{m_i(t)}{\sum_{j=1}^N m_j(t)},$$

где

$$m_i(t) = \frac{f(p_i) - \max_{j \in \{1, \dots, N\}} f(p_j)}{\min_{j \in \{1, \dots, N\}} f(p_j) - \max_{j \in \{1, \dots, N\}} f(p_j)}$$

Достоинства рассматриваемого метода: простота реализации; на практике метод точнее, чем генетические алгоритмы с вещественным кодированием и классический алгоритм роя частиц; большая скорость сходимости, чем у генетических алгоритмов с вещественным кодированием и классического алгоритм роя частиц.

К недостаткам рассматриваемого метода следует отнести следующие: не самая большая скорость за счет необходимости пересчёта многих параметров; большая часть достоинств теряется при оптимизации мультимодальных функций (особенно больших размерностей), так как метод начинает быстро сходиться к некоторому локальному оптимуму, из которого сложно выбраться, так как не предусмотрены процедуры, похожие на мутации в генетических алгоритмах.

Оригинальный алгоритм склонен к преждевременной сходимости в случае поиска экстремума сложных мультимодальных целевых функций. В связи с этим разработано значительное число модификаций алгоритма, имеющих целью преодоление данного его недостатка. Вариантом решения данной проблемы является введение в оригинальный алгоритм гравитационного поиска операторов, так называемых знаковой пунктуации и переупорядочивающей мутации.

Контрольные вопросы

- 1) Естественная мотивация роя частиц. Три основных принципа, которых должна придерживаться каждая отдельная частица.
- 2) Формальное описание метода роя частиц. Блок-схема алгоритма.
- 3) Вариации алгоритма роя частиц. Основные моменты изменения базового алгоритма.
- 4) Основные аспекты и параметры роевых алгоритмов. Критерии останова алгоритма.
- 5) Естественная мотивация алгоритма муравьиной колонии. Базовая идея алгоритма муравья. Формальное описание муравьиного алгоритма для задачи коммивояжёра.
- 6) Приведите пример расчета муравьиного алгоритма для одной итерации.
- 7) Приведите примеры модификаций муравьиного алгоритма.
- 8) Естественная мотивация алгоритма пчелиной колонии.
- 9) Формальное описание алгоритма пчелиной колонии применительно к нахождению экстремумов функции. Блок-схема алгоритма.
- 10) Пример расчета итерации алгоритма пчелиной колонии.
- 11) Естественная мотивация алгоритма светлячков. Формальное описание алгоритма, инспирированного поведением роя светлячков.
- 12) Естественная мотивация алгоритма поиска кукушки. Формальное описание алгоритма поиска кукушки. Псевдокод алгоритма.
- 13) Естественная мотивация алгоритма летучих мышей. Формальное описание алгоритма летучих мышей.
- 14) Естественная мотивация алгоритмов поведения обезьян – алгоритм обезьяньего поиска и обезьяний алгоритм. Формальное описание обезьяньего алгоритма.
- 15) Естественная мотивация тасующего алгоритма прыгающих лягушек. Формальное описание алгоритма прыгающих лягушек.
- 16) Естественная мотивация алгоритма поиска косяком рыб. Операторы алгоритма поиска косяком рыб.
- 17) Естественная мотивация тасующего алгоритма сорняковой оптимизации. Формальное описание алгоритма сорняковой оптимизации.
- 18) Естественная мотивация метода бактериальной оптимизации. Формальное описание алгоритма и его модификации.
- 19) Естественная мотивация алгоритма гравитационного поиска. Формальное описание алгоритма гравитационного поиска.

Список литературы

1. Карпенко А. П. Современные алгоритмы поисковой оптимизации. Алгоритмы, вдохновленные природой: учебное пособие / А. П. Карпенко. – Москва: Издательство МГТУ им. Н. Э. Баумана, 2014. – 448 с.
2. Кубил В.Н., Мохов В.А. К вопросу о применении роевого интеллекта в решении задач транспортной логистики // Проблемы модернизации инженерного образования в России / Юж.-Рос. гос. политехн. ун-т (НПИ) – Новочеркасск: ЮРГПУ(НПИ), 2014. – С. 140-144.
3. Баранюк В.В., Смирнова О.С. Роевой интеллект как одна из частей онтологической модели бионических технологий. *International Journal of Open Information Technologies*. Vol.3, no. 12, 2015. P.28 – 36.
4. Гринченков Д.В. Вариант реализации роевого алгоритма летучих мышей // Изв. вузов. Сев.-Кавк. регион. Техн. науки – 2015. – № 4. – С. 22 – 27.
5. Водолазский И.А., Егоров А.С., Краснов А.В., Терехов В.И. Роевой интеллект и его наиболее распространенные методы реализации // Молодой ученый. – 2007. – №4 (138). – С. 147 – 153.
6. Масимканова Ж.А. Обзор современных методов роевого интеллекта для компьютерного молекулярного дизайна лекарственных препаратов // Проблемы информатики – 2016. – №2 (31). – С. 50 – 61.
7. Agrawal S., Silakari S. A review on application of Particle Swarm Optimization in Bioinformatics // *Nurrent bioinformatics*. 2015. Vol. 10. P. 401-413.
8. M. Dorigo, L. M. Gambardella, Ant Colony System: A Cooperative Learning Approach to the Traveling Salesman Problem // *IEEE Transactions on Evolutionary Computation*. 1997. Vol. 1. P. 53-66.
9. Джонс М. Т. Программирование искусственного интеллекта в приложениях / М. Т. Джонс; Пер. с англ. Осипов А. И. – М.: ДМК Пресс, 2011. – 312 с.
10. Штовба С.Д. Муравьиные алгоритмы // *Exponenta Pro. Математика в приложениях*, 2003 г., № 4, с. 70–75.
11. D. T. Pham, A. Ghanbarzadeh, E. Koc, S. Otri, S. Rahim, M. Zaidi, The Bees Algorithm A Novel Tool for Complex Optimisation Problems, *Proceedings of IPROMS 2006 Conference*, P. 454 –461.
12. Кальчевская П. И., Леванова Т. В. Алгоритм пчелиного роя для задачи размещения предприятий с ограничениями на объемы поставок. Сибирская государственная автомобильно-дорожная академия (СибАДИ), 2015. С. 1833–1837.

13. Курейчик В. В., Жиленков М. А. Пчелиный алгоритм для решения оптимизационных задач с явно выраженной целевой функцией // Информатика, вычислительная техника и инженерное образование. 2015. Вып. 1, № 21. С. 1–8.
14. Teodorović D. Bee Colony Optimization (BCO) // Innovations in Swarm Intelligence. Springer, Berlin, Heidelberg, 2009. P. 39–60.
15. Pham D. T., Haj Darwish A., Eldukhri E. E. Optimisation of a fuzzy logic controller using the bees algorithm // Int. J. Comput. Aided Eng. Technol. 2009. V. 1, № 2. P. 250–264.
16. Karaboga D., Basturk B. On the performance of artificial bee colony (ABC) algorithm // Appl. Soft Comput. 2008. V. 8, № 1. P. 687–697.
17. В.В. Курейчик, Д.В. Заруба, Д.Ю. Запорожец Алгоритм параметрической оптимизации на основе модели поведения роя светлячков // Известия ЮФУ. Технические науки, 2015. с. 6 – 15.
18. Hongxia Liu, Shiliang Chen, Yongquan Zhou. A novel hybrid optimization algorithm based on glowworm swarm and fish school // Journal of Computational Information Systems. – 2010. – № 6 (13). – P. 4533 – 4542.
19. Piotr and Oramus. Improvements to glowworm swarm optimization algorithm // Computer Science. – 2010. – № 11. – P. 7 – 20.
20. Yang X.-S., S. Deb S. Cuckoo search via Lévy flights // In: Proc. Of World Congress on Nature & Biologically Inspired Computing (NaBIC 2009), 2009, India, P. 210 – 214.
21. Tuba M., Subotic M., Stanarevic N. Modified cuckoo search algorithm for unconstrained optimization problems // In: Proceedings of the European Computing Conference, 2010, P. 263 – 268.
22. Valian Eh., Mohanna Sh., Tavakoli S. Improved cuckoo search algorithm for feedforward neural network training // In: International Journal of Artificial Intelligence & Applications (IJAIA), 2011, Vol.2, No.3, pp. 36 – 43.
23. Yang X.-S. A New Metaheuristic Bat-Inspired Algorithm, in: Nature Inspired Cooperative Strategies for Optimization (NISCO 2010). Studies in Computational Intelligence. Berlin: Springer, 2010. Vol. 284. P.65 – 74.
24. Yang X. S. Bat algorithm for multi-objective optimization /X. S. Yang // International Journal of Bio-Inspired Computation. – 2011. – Vol. 3, № 5. – P. 267 – 274
25. Zhao R., Tang W. Monkey Algorithm for Global Numerical Optimization // Journal of Uncertain Systems. – 2008. – V.2, N.3, P.165 – 176.

26. Chen X., Zhou Y., Luo Q. A Hybrid Monkey Search Algorithm for Clustering Analysis // The Scientific World Journal – 2014. – Article ID 938239, 16 p.
27. И.А. Ходашинский, М.Б. Бардамова, В.С. Ковалев Отбор признаков и построение нечеткого классификатора на основе алгоритма прыгающих лягушек // Искусственный интеллект и принятие решений. 2018., № 1. С. 76 – 84.
28. Narimani M. R. A New Modified Shuffle Frog Leaping Algorithm for Non-Smooth Economic Dispatch // World Applied Sciences Journal. – 2011. – P. 803 – 814.
29. C.J.A.B Filho., F.B. de Lima Neto, A.J. C.C.. Lins, A.I.S. Nascimento., and M. P. Lima, A novel search algorithm based on fish school behavior // Systems, Man and Cybernetics, SMC 2008. IEEE International Conference on, 2008, P. 2646 – 2651.
30. Частикова В.А., Дружинина М.А., Кекало А.С. Исследование эффективности алгоритма поиска косяком рыб в задаче глобальной оптимизации // Современные проблемы науки и образования. – 2014, № 4 URL: <http://www.science-education.ru/ru/article/view?id=14142> (дата обращения: 07.01.2019)
31. Rad H.S., Lucas C. A Recommender System based in Invasive Weed Optimization Algorithm // IEEE Congress on Evolutionary Computation (CEC 2007). – 2007. – P. 4297 – 4304.
32. Passino K.M. Biomimicry of Bacterial Foraging for Distributed Optimization and Control // IEEE Control System Magazine. – 2002. – № 3 (22). – P. 52 – 67.
33. E. Rashedi, H. Nezamabadi-Pour, S. Saryazdi GSA: a gravitational search algorithm // Information sciences, – 2009. – Vol. 179, P. 2232 – 2248.
34. Hamed Sh.-H. The intelligent water drops algorithm: a nature-inspired swarm-based optimization algorithm // international Journal of Bio-inspired computation – 2009. – Vol.1, p.71 – 79.