Санкт-Петербургский государственный университет
Прикладная математика и информатика

Отчет по учебной практике

Реализация и визуализация некоторых алгоритмов численной оптимизации

Выполнил:
Петров Никита Денисович
группа 23.Б04-мм

Научный руководитель:
к. ф.-м. н., доцент
Шпилёв Пётр Валерьевич
Кафедра Статистического Моделирования

Санкт-Петербург
2023

Оглавление

1. Введение

В своей работе я разберу и реализую некоторые алгоритмы поисковой оптимизации: алгоритм с возвратом при неудачном шаге, метод имитации отжига, генетический алгоритм. Напишу программу на языке программирования Python для каждого алгоритма для решения задачи коммивояжёра, сравню время их работы на разных тестовых данных.

2. Алгоритм с возвратом при неудачном шаге

Описание

Алгоритм с возвратом при неудачном шаге — это метод решения задачи перебора всех возможных вариантов с последующим выбором оптимального решения.

Принцип работы алгоритма основывается на рекурсивном итеративном процессе, который проходит по всем возможным ветвям решения задачи. Основной идеей является перебор всех возможных решений путем последовательного выбора вариантов и проверки их на соответствие заданным условиям.

Математическое описание

Рассматривается следующая многомерная задача локальной безусловной оптимизации: найти минимум критерия оптимальности $f(x)$, определенного в n-мерном евклидовом пространстве $\mathbb{R}^n$,

$$\min_{x \in \mathbb{R}^n} f(x) = f(x^*) = f^* \tag{1}$$

При решении задачм алгоритм использует итерационную формулу вида:

$$x_{t+1} = x_t + \lambda_t v_t \tag{2}$$

где вектор $v_t = \frac{\Psi_t}{\|\Psi_t\|}$, $\Psi_t = (\psi_1^{(t)}, \psi_2^{(t)}, \ldots, \psi_n^{(t)})$ - реализация n-мерного случайного вектора, $\|*\|$ - некоторая векторная норма. Обычно в качестве координат вектора Ψ_t используют независимые случайные величины, равномерно распределенные в интервале $[-1; 1]$.

Свободными параметрами алгоритма являются начальная длина шага λ°, предельное число неудачных попыпок m (рекомендуемое значение равно $3n$), коэффициент уменьшения шага $\nu \in (0; 1)$. Схема алгоритма имеет следующий вид.

1. Задаем начальную точку x°, начальные значения параметров λ°, m, ν и полагаем счетчик числа итераций $t = 0$.

2. Задаем начальное значение счетчика числа неудачных попыток $k = 1$.

3. Получаем реализацию вектора случайных чисел $\Psi_t = (\psi_1^{(t)}, \psi_2^{(t)}, \ldots, \psi_n^{(t)})$, по формуле (2) находим пробную точку x_{t+1} и вычисляем величину $f(x_{t+1})$

4. Если $f(x_{t+1}) < f(x_t)$, то полагаем $t = t+1$ и переходим к шагу 2. Иначе - переходим к шагу 5.

5. Если $k < m$,тополагаем $k = k + 1$ иперехoдим к шагу 3. Иначе - переходим к шагу 6.

6. Проверяем условие окончания поиска. Если это условие выполнено, то полагаем $x^* \approx x^{t+1}$ и завершаем итерации, иначе полагаем $t = t + 1$, $\lambda_t = \nu\lambda_t$ и переходим к шагу 2.

В качестве условия окончания итераций может быть использовано одно из стандартных условий окончания итераций:

$$\|x_{t+1} - x_t\| = \lambda_t \leq \epsilon_x, \tag{3}$$

где ϵ_x - константа, определяющаю требуемую точность решения по x.

3. Метод имитации отжига

Описание

Алгоритм основывается на имитации физического процесса, который происходит при отжиге металлов. Процесс заключается в том, чтобы сначала сильно нагреть металл, а затем медленно снижать его температуру. Сначала атомы двигаются очень быстро и с какой-то вероятностью могут поменять свое место в решетке. Когда начинается остывание эта вероятность понижается вместе с понижением температуры. В конечном счете образуется устойчивая кристаллическая решетка, соответствующая минимуму энергии атомов.

Первым делом задается начальное состояние. Затем производится небольшое изменение этого состояния. Если новое состояние лучше прежнего, то теперь оно становится начальным, иначе мы не отбрасываем его, а с какой-то вероятностью, которая зависит от температуры. Чем выше температура, тем ваше вероятность. Эти шаги повторяются пока не выполнено какое-то количество итераций или не будет достигнут желаемый результат.

Математическое описание

Как в алгоритме с возвратом при неудачном шаге у нас есть целевая функция $f : \mathbb{R}^n \to \mathbb{R}$, у которой мы хотим найти минимум. То есть пара $x^*, f(x^*)$ из формулы (1) будет искомым решением.

1. Сперва зададим следующие параметры:
   - T - начальная температура
   - Количество итераций, которое будет сделаны алгоритмом в поиске решения. Обозначим это количество как l.
   - Если заранее известно минимальное значение функции, то его тоже необходимо задать.
2. Задаем начальное состояние $x \in \mathbb{R}^n$
3. В небольшой окрестности $\mathring{U}(x)$ точки x выбираем случайное решение x', не сильно отличающееся от вектора x.

4. Если $\Delta f = f(x') - f(x) < 0$, то в качестве нового текущего приближения принимаем x'. Если же $\Delta f = f(x') - f(x) > 0$, то мы не отбрасываем решение x' полностью, а принимаем с вероятностью $P = \exp^{-\frac{\Delta f}{T}}$.

5. Если $\Delta f = f(x') - f(x) > 0$, то изменяем T по какому-то закону. Например, $T_m = \frac{T_0}{\ln(m+1)}$, T_0 - начальная температура. Последовательность $\{T_n\}_{n \in \mathbb{N}}$ обязательно должна быть убывающей, что характерезует постепенное остывание металла.

6. Увеличим счетчик итераций на один: $k = k + 1$. Если $k < l$, то переходим на следующую итерацию. Иначе выходим из цикла и получаем текущее состояние оптимальное решение $(x^*, f(x^*))$. Но возможно за данное количество итераций алгоритм не найдет самое оптимальное решение.

4. Генетический алгоритм

Описание

Генетический алгоритм - это эволюционный алгоритм поиска. Он используется для решения задач оптимизации и моделирования путём случайного подбора, комбинирования и вариации искомых параметров с использованием механизмов, аналогичных естественному отбору в природе.

Схема канонического генетического алгоритма имеет следующий вид:

1. Агентов (особей, индивидуумов) представляем в виде хромосом.
2. Случайным образом создаем некоторое число исходных особей - *начальную популяцию.*
3. Особи оцениваем с помощью фитнес-функции - каждой особи ставим в соответствие определенное значение *приспособленности*, которое определяет вероятность ее выживания.
4. На основе приспособленностей выбираем особи для скрещивания - этап *селекции.* К хромосомам этих особей применяем *генетические операторы* скрещивания и мутации, создавая таким образом следующее *поколение* особей.
5. Особи созданного поколения также оцениваем, производим селекцию, применяем генетические операторы и так далее до тех пор, пока не будет вы полнен критерий останова алгоритма.

Суть эволюционных алгоритмов, как и популяционных алгоритмов в целом, состоит в обеспечении более высокой средней приспособленности нового поколения по сравнению с такой же приспособленностью предыдущего поколения.

Известно очень большое число вариантов рассмотренной схемы генетического алгоритма, отличающихся способами кодирования хромосом, набором генетических операторов, структурой и параметрами алгоритма и т. д.

Математическое описание

Пусть $f : \mathbb{R}^n \to \mathbb{R}$ - целевая функция(фитнесс-функция), у которой мы хотим найти максимум. То есть пара $x^*, f(x^*)$ из формулы (1) будет искомым решением.

Генетический алгоритм требует определения следующих параметров:

- **Размер популяции** - количество особей, которые будут находиться в популяции на каждой итерации. Это число постоянно и не будет меняться в процессе поиска решения. Обозначим это число как size.
- **Вероятность мутации** - вероятность, с которой новые особи будут претерпевать изменения генома, т.е. мутировать. Обозначим это число как ??
- **Размер хромосом особей** - количество параметров фитнесс-функции, которую мы хотим оптимизировать. Это число постоянно и не будет меняться в процессе поиска решения. Обозначим это число как n.
- **Количество поколений поиска** - число поколений, которое будет сгенерировано в результате поиска. Обозначим это число как L.
- Если заранее известно минимальное значение функции, то его тоже необходимо задать. Обозначим его как optimalf.

Шаги генетического алгоритма:

1. **Создание начальной популяции**. Представим каждую особь A в виде хромосомы $H = H_i : i \in [1 : n]$, где H_i - код компоненты x_i вектора x. Легко видеть, что имеет место соотношение:
$$\sum_{i=1}^{n} |H_i| = |H|$$
Таким образом каждая особь задается как $A = \langle x, H, f(x) \rangle$. Где x - фенотип особи, H - генотип, $f(x)$ - приспособленность особи.
Пусть $S = \{A_1, A_2, \ldots, A_{size}\}$ - начальная популяция.
2. **Отбор(селекция)**. Пусть $P = \{p_1, p_2, \ldots, p_{size}\}$ - приспособленность каждой особи. Тогда для отбора будем использовать метод рулетки. То есть вероятность выбора особи тем вероятнее, чем лучше её значение функции приспособленности
$$\varphi_i = \frac{p_i}{\sum_{i=1}^{size} p_i}$$
Выбираем $\frac{size}{2}$ пар особей (A_i, A_j) из популяции $S = \{A_1, A_2, \ldots, A_{size}\}$ с вероятностями φ_i и φ_j соответственно.

3. **Размножение(скрещевание)**. Теперь скрещеваем особи из пары между собой. Пусть $q \in [1, n-1]$ - разделитель, по которому будут разделяться хромосомы особей. Он может быть фиксированным числом или задаваться случайно. Если $A_i = (H_1^{(i)}, \ldots, H_n^{(i)})$, $A_j = (H_1^{(j)}, \ldots, H_n^{(j)})$, то в результате скрещевания у нас получаться два потомка: $C_i = (H_1^{(i)}, \ldots, H_q^{(i)}, H_{q+1}^{(j)}, \ldots, H_n^{(j)})$, $C_j = (H_1^{(j)}, \ldots, H_q^{(j)}, H_{q+1}^{(i)}, \ldots, H_n^{(i)})$. Получим совокупность $C = \{C_1, \ldots, C_{size}\}$ - потомство популяции $S = \{A_1, A_2, \ldots, A_{size}\}$.

4. **Мутация**. Преобразуем всех потомков с какой-то вероятностью р. $C_i = (H_1^{(i)}, \ldots, H_n^{(i)})$ в $C_i = (H_1^{(i)}, \ldots, \hat{H}_q^{(i)}, \ldots, H_n^{(i)})$, где $q \in [1 : n]$ выбирается случайно.

5. **Замена начальной популяции**. Заменяем всех предков на их потомков. Теперь $S = \{C_1, C_2, \ldots, C_{size}\}$.

6. **Проверка условия прекращения поиска**. Выбираем $C_i \in S$ - лучшую особь. Если $f(C_i) = min_f$ или $k \leq L$, то прекращаем поиск. Иначе увеличиваем $k = k+1$ и повторяем шаги 2-5.

5. Задача коммивояжёра

Описание

Задача заключается в том, чтобы найти самый короткий путь, проходящий по одному разу через указанные города с возвратом в исходный город.

Математическое описание

Пусть $f : \mathbb{Z}^n \to \mathbb{R}$ функция, которая подсчитывает общее расстояние между городами. Тогда чтобы решить задачу нам нужно найти $\min\{f(x) : x \in \mathbb{Z}^n\} = f(x^*)$. Соответственно пара $(x^*, f(x^*))$ - решение задачи. Очевидно, что если x^* - решение, то $f(x^*) = 0$.

Начальное состояние представлено следующим образом: $x = (x_1, x_2, \ldots, x_n) \in \mathbb{Z}^n$. Такое представление означает путь $x_1 \to x_2 \to \ldots \to x_n \to x_1$, где x_i - города. Тогда нам необходимо найти $x^* = \sigma \in S_n : f(x^*) = 0$. Данный принцип соблюдается в трех алгоритмах.

Алгоритм с возвратом при неудачном шаге

Начальной точкой является тождественная перестновка, то есть $x^\circ = (1, 2, \ldots, n)$, далее идет перебор всех перестановок x°. Соответственно окончанием итераций является конец перебора всех перестановок.

Ниже представлены времы работы алгоритма для разных n. При больших n приходится очень долго ждать результата.

Size	n = 3	n = 5	n = 7	n = 9	n = 11
Time, c	5.61e-6	5.36e-5	2.25e-3	0.18	23.06

Метод имитации отжига

Пусть начальные параметры будет следующие:

1. $T = 5000$
2. $\alpha = 0.999$
3. Начальная точка $x = (1, 2, \ldots, n)$

Соответственно температура будет изменяться по формуле $T_i = T_{i-1}\alpha$

Ниже представлены времы работы алгоритма для разных n.

Size	n = 5	n = 50	n = 100	n = 500	n = 2000
Time, c	0.0018	0.0078	0.015	0.087	0.4

Генетический алгоритм

Особями будут выступать перестановки n-элементного множества, поэтому нет смысла двоичного кодирования, как это должно происходить в каноническом генетическом алгоритме.

Оператор мутации меняет местами два элементв особи. Вероятность мутации $\mu = 0.25$.Результат работы для разных n предаставлен ниже.

Size	n = 10	n = 50	n = 100	n = 200	n = 300
Time, c	0.93	2.9	6.05	29.2	90.95
Population size	20	30	30	40	50

6. Заключение

В своей работе я рассмотрел три алгоритма поисковой оптимизации, реализовал их на языке программирования Python для решения задачи коммивояжера. Также довольно наглядно показано сравнение времени работы этих алгоритмов.

Самым простым в плане реализации оказался алгоритм с возвратом при неудачном шаге, но в то же время он оказался самым слабым в плане решения задачи.

Генетический же алгоритм был самым сложным в реализации, а результаты его работы оказались средними.

Метод имитации отжига реализовывается довольно просто и очень быстро может решить задачу, что делает его самым эффективным из этих трех алгоритмов.

7. Список литературы

1. Современные алгоритмы поисковой оптимизации. Алгоритмы, вдохновленные природой : учебное пособие / А. П. Карпенко. — Москва : Издательство МГТУ им. Н. Э. Баумана, 2014 — 446, [2] с. : ил. ISBN 978-5-7038-3949-2

2. https://ru.wikipedia.org/wiki/Генетический алгоритм

3. http://bigor.bmstu.ru/?cnt/?doc=MO/ch0801.mod/?cou=MO/base.cou

Приложение

Алгоритм с возвратом при неудачном шаге

```
import itertools


def f(x, graph):
'''Функция, вычисляющаю длину маршрута'''
        distance = 0
        for i in range(len(x) - 1):
                distance += graph[x[i]][x[i + 1]]
        distance += graph[x[0]][x[-1]]
        return distance


def min_distance(initial_x, graph):
'''Функция, вычисляющаю минимальную длину маршрута'''
        current_f = f(initial_x, graph)
        final_x = initial_x
        for current_x in itertools.permutations(initial_x):
                next_f = f(current_x, graph)
                if next_f < current_f:
                        current_f = next_f
                        final_x = current_x
        final_f = current_f
        return final_f
```

Метод имитации отжига

```
import random


def f(x, graph):
'''Функция, вычисляющаю длину маршрута'''
        distance = 0
        for i in range(len(x) - 1):
                distance += graph[x[i]][x[i + 1]]
        distance += graph[x[0]][x[-1]]
        return distance



def shuffle(old_state, n):
'''Производим перестановку двух случайных координат вектора'''
        first_coordinate = random.randint(0, n - 1)
        second_coordinate = random.randint(0, n - 1)

        new_state = old_state[::]
        new_state[first_coordinate] = old_state[second_coordinate]
        new_state[second_coordinate] = old_state[first_coordinate]

        return new_state



def SimAnnealing(graph, n, x):
'''Метод имитации отжига'''
        T = 5000
        a = 0.999
        limit = 1000
        count = 0

        while (count < limit):
```

```
                new_state = shuffle(x, n)
                if (f(new_state, graph) - f(x, graph)) < 0:
                        x = new_state
                else:
                        diff_f = f(new_state, graph) - f(x, graph)
                        p = math.e ** (-(diff_f / T))
                        if random.random() <= p:
                                x = new_state
                                T *= a
                count += 1
        return f(x, graph)
```

Генетический алгоритм

```
import random


def f(x, graph):
'''Функция, вычисляющаю длину маршрута'''
        sum_distance = 0
        for i in range(len(x) - 1):
                sum_distance += graph[x[i]][x[i + 1]]
        sum_distance += graph[x[0]][x[-1]]
        return sum_distance


def probability(number, start, end):
'''Вероятность'''
        random_number = random.uniform(start, end)
        if random_number < number:
                return True
        else:
                return False


def parents_choice(population, graph):
```

```
    '''Выбираем двух родителей в соответствии с приспособленностью'''
        p = P(population, graph)
        s = 0
        for i in range(len(p)):
            s += p[i]
            if probability(s, 0, len(population) - 1):
                first_parent = population[i]
                break

        s = 0
        for i in range(len(p)):
            s += p[i]
            if probability(s, 0, len(population) - 1):
                second_parent = population[i]
        break

        parents = [first_parent, second_parent]

        return parents

def mutation(individual):
    '''Производим перестановку двух хромосом у особи'''
        first_individual_c = random.randint(0, len(individual) - 1)
        second_c = random.randint(0, len(individual) - 1)
        chromosome_individual = individual[first_individual_c]
        individual[first_individual_c] = individual[second_c]
        individual[second_c] = chromosome_individual

def crossing(first_parent, second_parent):
    '''Производим скрещевание двух родителей, получая потомство'''
        q = random.randint(1, len(first_parent) - 1)
        first_descendant = first_parent[:q]
```

```
        for i in second_parent[q:]:
                if not (i in first_descendant):
                        first_descendant.append(i)

        second_descendant = second_parent[:q]
        for i in first_parent[q:]:
                if not (i in second_descendant):
                        second_descendant.append(i)
        descendants = [first_descendant +
[i for i in first_parent if not(i in first_descendant)],
second_descendant +
[i for i in second_parent if not(i in second_descendant)]]

        return descendants


def P(population, graph):
'''Считаем приспособленность особи'''
p = [
(1 - (f(population[i], graph) / sum([f(population[j], graph) for j in
range(len(population))]))) for i in range(len(population))
]

return p


def new_population(population, probability_mutation):
'''Создаем новую популяцию'''
        new_population = []

        for i in range(int(len(population) / 2)):
                parents = parents_choice(population, graph)
```

```
                descendants = crossing(parents[0], parents[1])
                new_population.append(descendants[0])
                new_population.append(descendants[1])

        for i in range(len(new_population)):
                if probability(probability_mutation, 0, 1):
                        mutation(new_population[i])
                else:
                        mutation(new_population[i])

        return new_population

def GeneticAlgorithm(population, graph, optimal_f, limit):
'''Генетический алгоритм'''
        count = 0
        probability_mutation = 0.25
        min_f = min([f(individ, graph) for individ in population])
        while (count <= limit) or (optimal_f > min_f):
                population = new_population(population,
                                            probability_mutation)
                min_f = min([f(individ, graph)
                             for individ in population])
                count += 1

        return min_f
```