НЕКОТОРЫЕ ПРАКТИЧЕСКИЕ ЗАДАЧИ ПРИМЕНЕНИЯ ГЕНЕТИЧЕСКИХ АЛГОРИТМОВ

Вирсански Э. Генетические алгоритмы на Python / пер. с англ. А. А. Слинкина. – М.: ДМК Пресс, 2020. – 286 с.: ил.

Каркас DEAP

Для работы с генетическими алгоритмами создан целый ряд каркасов на Python, например GAFT, Pyevolve и PyGMO. Но, ноиболее привлекательным прдеставляется каркас **DEAP**, поскольку он прост в использовании и предлагает широкий набор функций, поддерживает расширяемость и может похвастаться подробной документацией.

DEAP (сокращение от Distributed Evolutionary Algorithms in Python – распределенные эволюционные алгоритмы на Python) поддерживает быструю разработку решений с применением генетических алгоритмов и других методов эволюционных вычислений.

DEAP предлагает различные структуры данных и инструменты, необходимые для реализации самых разных решений на основе генетических алгоритмов. Каркас DEAP был разработан в канадском университете Лаваля в 2009 г. и предлагается на условиях лицензии GNU Lesser General Public License (LGPL).

Исходный код DEAP доступен по адресу https://github.com/DEAP/deap, а документация размещена по адресу https://deap.readthedocs.io/en/master//

Использование модуля CREATOR

DEAP creator используется как метафабрика и позволяет расширять существующие классы, добавляя в них новые атрибуты. Пусть, например, имеется класс **Employee.** С помощью модуля **creator** мы можем создать из него **класс Developer** следующим образом

```
from deap import creator
creator.create("Developer", Employee, position = "Developer",
programmingLanguages = set)
```

Первым аргументом функции create() передается имя нового класса, вторым – существующий класс, подлежащий расширению. Все последующие аргументы определяют атрибуты нового класса. Если значением аргумента является класс (например, dict или set), то он будет добавлен в новый класс как атрибут экземпляра, инициализируемый в конструкторе. Если же это не класс (а, например, литерал), то он добавляется как атрибут класса (статический).

Таким образом, созданный класс Developer расширяет класс Employee и имеет атрибут класса position, равный строке Developer, и атрибут экземпляра programmingLanguages типа set, который инициализируется в конструкторе. Следовательно, новый класс эквивалентен такому:

```
class Developer(Employee):
        position = "Developer«
        def __init__(self):
                self.programmingLanguages = set()
```

При работе с DEAP модуль **creator** обычно служит для создания классов **Fitness** и **Individual**, используемых в генетических алгоритмах.

Создание класса Fitness

При работе с DEAP значения приспособленности инкапсулированы в классе Fitness. DEAP позволяет распределять приспособленность по нескольким компонентам (называемым целями), у каждого из которых есть свой вес. Комбинация весов определяет поведение, или стратегию приспособления в конкретной задаче.

Определение стратегии приспособления.

Для определения стратегии в состав DEAP входит абстрактный класс base. Fitness, который содержит кортеж weights. Этому кортежу необходимо при своить значения, чтобы определить стратегию и сделать класс пригодным для использования. Для этого мы расширяем базовый класс Fitness с помощью модуля creator так же, как делали это ранее с классом Developer:

```
creator.create("FitnessMax", base.Fitness, weights=(1.0,))
```

Получается класс creator.FitnessMax, расширяющий класс base.Fitness, в котором атрибут класса weights инициализирован значением (1.0,).

**Обратите внимание на запятую в конце определения weights, которая присутствует, хотя задан всего один вес. Она необходима, потому что weights – кортеж.*

Стратегия этого класса FitnessMax – **максимизировать приспособленность** индивидуумов с единственной целью. Если бы нам нужно было минимизировать приспособленность в задаче с одной целью, то для задания соответствующей стратегии можно было бы определить такой класс:

```
creator.create("FitnessMin", base.Fitness, weights=(-1.0,))
```

Можно также определить класс для оптимизации сразу нескольких целей различной важности:

```
creator.create("FitnessCompound", base.Fitness, weights=(1.0, 0.2, -0.5))
```

В классе creator.FitnessCompound используется три компоненты приспособленности с весами 1.0, 0.2 и –0.5. Это означает, что первая и вторая компоненты (цели) максимизируются, а третья минимизируется, причем первая компонента самая важная, следующей по важности является третья, а последней – вторая.

Хранение значения приспособленности

Если кортеж weights определяет стратегию приспособления, то кортеж values используется для хранения самих значений функции приспособленности в базовом классе base.Fitness. Эти значения дает отдельно определяемая функция, которую обычно называют evaluate(); мы опишем ее ниже в этой главе. Как и weights, кортеж values содержит по одному значению для каждой компоненты приспособленности (цели).

Третий кортеж, wvalues, содержит взвешенные значения, полученные перемножением элементов кортежа values и соответственных элементов кортежа weights. Всякий раз, как устанавливаются значения приспособленности, соответствующие взвешенные значения вычисляются и сохраняются в wvalues. Они используются при сравнении индивидуумов.

Взвешенные приспособленности можно сравнивать лексикографически с помощью следующих операторов:

>, =, <=, ==, !=

Созданный класс Fitness будет использован в определении класса Individual, как описано в следующем разделе.

Создание класса Individual

Второе типичное применение модуля **creator** – определение индивидуумов, образующих популяцию в генетическом алгоритме. В предыдущих главах мы видели, что индивидуумы представлены хромосомами, которыми можно манипулировать с помощью генетических операторов. В DEAP класс Individual создается путем расширения базового класса, представляющего хромосому. Кроме того, каждый экземпляр класса Individual должен содержать функцию приспособленности в качестве атрибута.

Чтобы удовлетворить обоим требованиям, мы воспользуемся модулем creator для создания класса creator.Individual:

```
creator.create("Individual", list, fitness=creator.FitnessMax)
```

Результат работы этой строки двоякий:

- созданный класс Individual расширяет встроенный класс Python list.
  Это означает, что все хромосомы имеют тип list;

- в каждом экземпляре класса Individual имеется атрибут fitness созданного ранее класса FitnessMax.

Использование класса Toolbox

Второй механизм, предлагаемый каркасом **DEAP**, – класс base.Toolbox. Он используется как контейнер для функций (или операторов) и позволяет создавать новые операторы путем назначения псевдонимов или настройки существующих функций.

Пусть, например, имеется следующая функция sumOfTwo():

```
def sumOfTwo(a, b):
 return a + b
```

Воспользовавшись модулем toolbox, мы сможем создать новый оператор incrementByFive() на основе функции sumOfTwo():

```
from deap import base
toolbox= base.Toolbox()
toolbox.register("incrementByFive", sumOfTwo, b=5)
```

Первым аргументом функции register() передается имя нового оператора (или псевдоним существующего), вторым – настраиваемая функция. Все остальные (необязательные) аргументы передаются этой функции при вызове нового оператора. Рассмотрим, к примеру, следующее определение:

```
toolbox.incrementByFive(10)
```

Этот вызов эквивалентен такому: `sumOfTwo(10, 5)`

поскольку аргументу b было присвоено фиксированное значение 5 в определении оператора incrementByFive.

Создание генетических операторов

Во многих случаях класс Toolbox используется для настройки существующих функций из модуля tools, который содержит ряд полезных функций, относящихся к генетическим операциям отбора, скрещивания и мутации, а также утилиты для инициализации.
Например, в коде ниже определены три псевдонима, которые впоследствии будут использованы как генетические операторы:

```
rom deap import tools
toolbox.register("select", tools.selTournament, tournsize=3)
toolbox.register("mate", tools.cxTwoPoint)
toolbox.register("mutate", tools.mutFlipBit, indpb=0.02)
```

Опишем, что здесь было сделано.

- Имя select зарегистрировано как псевдоним существующей в модуле tools функции selTournament() с аргументом tournsize = 3. В результате создается оператор toolbox.select, который выполняет турнирный отбор с размером турнира 3.
- Имя mate зарегистрировано как псевдоним существующей в модуле tools функции cxTwoPoint(). В результате создается оператор toolbox. mate, который выполняет двухточечное скрещивание.
- Имя mutate зарегистрировано как псевдоним существующей в модуле tools функции mutFlipBit с аргументом indpb = 0.02. В результате создается оператор toolbox.mutate, который выполняет мутацию инвертированием бита с вероятностью 0.02.

Модуль tools предоставляет реализации различных генетических операторов,

Функции отбора находятся в файле selection.py. Перечислим некоторые из них:

- selRoulette() – отбор по правилу рулетки;
- selStochasticUniversalSampling() – стохастическая универсальная выборка;
- selTournament() – турнирный отбор. Функции скрещивания находятся в файле crossover.py:
- cxOnePoint() – одноточечное скрещивание;
- cxUniform() – равномерное скрещивание;
- cxOrdered() – упорядоченное скрещивание (OX1);
- cxPartialyMatched() – скрещивание с частичным сопоставлением (partially matched crossover – PMX).

В файле mutation.py находятся две функции мутации:

- mutFlipBit() – мутация инвертированием бита;
- mutGaussian() – нормально распределенная мутация.

Создание популяции

Файл init.py модуля tools содержит несколько функций, полезных для создания и инициализации популяции. Особенно полезна функция initRepeat(), принимающая три аргумента:

- тип контейнера результирующих объектов;
- функция, генерирующая объекты, которые помещаются в контейнер;
- сколько объектов генерировать.

Например, следующая строка создает список из 30 случайных чисел от 0 до 1:

```
randomList = tools.initRepeat(list, random.random, 30)
```

В этом примере list – тип, выступающий в роли заполняемого контейнера, random.random – порождающая функция, а 30 – количество вызовов этой функции, необходимых для заполнения контейнера.

Что, если мы захотим заполнить список случайными целыми числами, равными 0 или 1? Можно было бы создать функцию, которая вызывает random.radint() для генерации одного случайного числа, равного 0 или 1, а затем использовать ее в качестве порождающей функции в initRepeat(), как показано ниже:

```
def zeroOrOne():
        return random.randint(0, 1)

randomList = tools.initRepeat(list, zeroOrOne, 30)
```

Или можно воспользоваться модулем toolbox:

```
toolbox.register("zeroOrOne", random.randint, 0, 1)
randomList = tools.initRepeat(list, toolbox.zeroOrOne, 30)
```

Здесь вместо того чтобы явно определять функцию zeroOrOne(), мы создали оператор (или псевдоним) zeroOrOne, который вызывает random.radint() с фиксированными параметрами 0 и 1.

Вычисление приспособленности

Как было отмечено выше, в классе Fitness задаются веса, определяющие стратегию приспособления (например, максимизация или минимизация), а сами значения приспособленности возвращает отдельно определенная функция. Эта функция обычно регистрируется в модуле toolbox под псевдонимом evaluate, как показано ниже:

```
def someFitnessCalculationFunction(individual):
        return _some_calculation_of_the_fitness

toolbox.register("evaluate",someFitnessCalculationFunction)
```

В данном примере функция someFitnessCalculationFunction() вычисляет приспособленность заданного индивидуума, а имя evaluate зарегистрировано в качестве ее псевдонима. Вот теперь мы готовы воспользоваться полученными знаниями и решить первую задачу, применив DEAP для реализации генетического алгоритма.

Задача OneMax OneMax (или One-Max)

Задача OneMax OneMax (или One-Max) – это простая задача оптимизации, которую часто приводят в пример как аналог программы «Hello World» в мире генетических алгоритмов.

Пример использования для демонстрации возможностей каркаса DEAP.

Задача OneMax состоит в том, чтобы найти двоичную строку заданной длины, для которой сумма составляющих ее цифр максимальна. Например, при решении задачи OneMax длины 5 будут рассматриваться такие кандидаты:

10010 (сумма цифр = 2);
11100 (сумма цифр = 3);
11111 (сумма цифр = 5).

Очевидно, что решением всегда является строка, состоящая из одних единиц.
Но генетический алгоритм не обладает таким знанием, поэтому должен слепо искать решение, пользуясь генетическими операторами. Если алгоритм справится с работой, то найдет решение (или приближение к нему) за разумное время.
В документации по каркасу DEAP задача OneMax приведена в качестве вводного примера (*https://github.com/DEAP/deap/blob/master/examples/ga/onemax.py*).

Выше перечислены несколько действий, которые нужно предпринять для решения задачи с помощью генетического алгоритма. Эти действия ниже будут оформлены в виде шагов.

Выбор хромосомы

Поскольку в задаче OneMax мы имеем дело с двоичными строками, с выбором хромосом проблем не возникает – каждый индивидуум представлен двоичной строкой, которая непосредственно соответствует потенциальному решению. На языке Python это реализуется в виде списка, содержащего числа 0 или 1. Длина хромосомы совпадает с размером задачи OneMax. Например, в задаче OneMax размера 5 индивидуум 10010 будет представлен списком [1, 0, 0, 1, 0].

Вычисление приспособленности

Поскольку мы хотим найти индивидуума с **наибольшей** суммой цифр, следует использовать стратегию FitnessMax. А поскольку каждый индивидуум представлен списком целых чисел, равных 0 или 1, то приспособленность вычисляется просто как сумма элементов списка, например: sum([1, 0, 0, 1, 0]) = 2.

Выбор генетических операторов

Теперь нужно решить, какие реализации генетических операторов отбора, скрещивания и мутации использовать. В предыдущей главе мы рассмотрели несколько таких реализаций. Выбор конкретного оператора – не точная наука, обычно приходится экспериментировать. Но если операторы отбора обычно могут работать с хромосомами любого типа, то операторы скрещивания и мутации должны соответствовать типу хромосомы, иначе будут получаться недопустимые хромосомы. В качестве оператора отбора можно для начала взять турнир, потому что его реализация проста и эффективна. Можно также будет поэкспериментировать с другими стратегиями, например правилом рулетки и SUS. Что касается скрещивания, то подойдет одноточечный или двухточечный оператор, поскольку в результате скрещивания двух двоичных строк этими методами получается допустимая двоичная строка. В качестве операции мутации можно взять простое инвертирование бита, этот метод хорошо работает для двоичных строк.

Задание условия остановки

Всегда имеет смысл ограничивать количество поколений, чтобы алгоритм не работал вечно. Тем самым мы получаем одно условие остановки. Кроме того, так уж получилось, что мы знаем наилучшее решение в задаче OneMax – двоичная строка из одних единиц со значением приспособленности, равным количеству единиц, – поэтому можно взять это в качестве второго условия остановки.

**Отметим, что в реальной задаче такая априорная информация обычно неизвестна.*

Если хотя бы одно условие выполнено – достигнут предел количества поколений или найдено наилучшее решение, то алгоритм останавливается.

Реализация средствами DEAP

Теперь можно, наконец, приступить к написанию кода решения задачи OneMax с помощью каркаса DEAP. Полный код программы имеется по адресу https://github.com/PacktPublishing/Hands-On-Genetic-Algorithms-with-Python/blob/master/Chapter03/01-OneMax-long.py

Подготовка

Прежде чем запускать главный цикл алгоритма, нужно все подготовить, и в DEAP это делается вполне определенным образом.

1. Сначала импортируем необходимые модули из DEAP и еще несколько вспомогательных библиотек:

```
from deap import base
from deap import creator
from deap import tools
import random
import matplotlib.pyplot as plt
```

2. Затем объявляем несколько констант, содержащих значения параметров самой задачи и генетического алгоритма:

```
# константы задачи
 ONE_MAX_LENGTH = 100        # длина подлежащей оптимизации битовой строки
# константы генетического алгоритма
 POPULATION_SIZE = 200       # количество индивидуумов в популяции
P_CROSSOVER = 0.9            # вероятность скрещивания
P_MUTATION = 0.1             # вероятность мутации индивидуума
MAX_GENERATIONS = 50         # максимальное количество поколений
```

3. Важный аспект генетического алгоритма – его вероятностный характер, поэтому в алгоритм нужно внести элемент случайности. Однако на этапе экспериментирования требуется, чтобы результаты были воспроизводимы. Для этого мы задаем какое-нибудь фиксированное начальное значение генератора случайных чисел:

```
RANDOM_SEED = 42
random.seed(RANDOM_SEED)
```

последствии эти строки нужно будет удалить, чтобы при разных прогонах получались разные результаты.

4. Выше мы видели, что одним из основных компонентов каркаса DEAP является класс Toolbox, который позволяет регистрировать новые функции (или операторы), настраивая поведение существующих функций. В данном случае мы воспользуемся им, чтобы определить оператор zeroOrOne путем специализации функции random.randomint(a, b). Эта функция возвращает случайное целое число N такое, что $a \leq N \leq b$. Если задать в качестве a и b фиксированные значения 0 и 1, то оператор zeroOrOne будет случайным образом возвращать 0 или 1. Во фрагменте ниже мы определяем переменную toolbox, а затем используем ее для регистрации оператора zeroOrOne:

```
toolbox = base.Toolbox()
toolbox.register("zeroOrOne", random.randint, 0, 1)
```

5. Далее следует создать класс Fitness. Поскольку у нас всего одна цель – сумма цифр, а наша задача – максимизировать ее, то выбираем стратегию FitnessMax, задав в кортеже weights всего один положительный вес:

```
creator.create("FitnessMax", base.Fitness, weights=(1.0,))
```

6. По соглашению, в DEAP для представления индивидуумов используется класс с именем Individual, для создания которого применяется модуль creator. В нашем случае базовым классом является list, т. е. хромосома представляется списком. Дополнительно в класс добавляется атрибут fitness, инициализируемый экземпляром определенного ранее класса FitnessMax:

```
creator.create("Individual", list, fitness=creator.FitnessMax)
```

7. Следующий номер нашей программы – регистрация оператора individualCreator, который создает экземпляр класса Individual, заполненный случайными значениями 0 или 1. Для этого мы настроим ранее определенный оператор zeroOrOne. В качестве базового класса используется вышеупомянутый оператор initRepeat, специализированный следующими аргументами:

- класс Individual в качестве типа контейнера, в который помещаются созданные объекты;
- оператор zeroOrOne в качестве функции генерации объектов;
- константа ONE_MAX_LENGTH в качестве количества генерируемых объектов (сейчас она равна 100).

Поскольку оператор zeroOrOne создает объекты, принимающие случайное значение 0 или 1, то получающийся в результате оператор individualCreator заполняет экземпляр Individual 100 случайными значениями 0 или 1:

```
toolbox.register("individualCreator", tools.initRepeat,
creator.Individual, toolbox.zeroOrOne, ONE_MAX_LENGTH)
```

8. Наконец, регистрируем оператор populationCreator, создающий список индивидуумов. В его определении также используется оператор initRepeat со следующими аргументами:

- класс list в качестве типа контейнера;
- оператор individualCreator, определенный ранее в качестве функции, генерирующей объекты в списке.

Последний аргумент initRepeat – количество генерируемых объектов – здесь не задан. Это означает, что при использовании оператора populationCreator мы должны будем указать этот аргумент, т. е. задать размер популяции:

```
toolbox.register("populationCreator", tools.initRepeat,
list, toolbox.individualCreator)
```

9. Для вычисления приспособленности мы сначала определим свободную функцию, которая принимает экземпляр класса Individual и возвращает его приспособленность. В данном случае мы назвали функцию, вычисляющую количество единиц в индивидууме, oneMaxFitness. Поскольку индивидуум представляет собой не что иное, как список значений 0 и 1, то на поставленный вопрос в точности отвечает встроенная функция Python sum():

```
def oneMaxFitness(individual):
        return sum(individual),      # вернуть кортеж
```

**Как уже было сказано, значения приспособленности в DEAP представлены кортежами, поэтому если возвращается всего одно значение, то после него нужно поставить запятую.*

10. Теперь определим оператор evaluate – псевдоним только что определенной функции oneMaxfitness(). Ниже мы узнаем, что использование псевдонима evaluate для вычисления приспособленности – принятое в DEAP соглашение:

```
toolbox.register("evaluate", oneMaxFitness)
```

11. Ранее мы говорили, что генетические операторы обычно создаются как псевдонимы существующих функций из модуля tools с конкретными значениями аргументов. В данном случае аргументы будут такими:

- турнирный отбор с размером турнира 3;
- одноточечное скрещивание;
- мутация инвертированием бита.

Обратите внимание на параметр indpb функции mutFlipBit. Эта функция обходит все атрибуты индивидуума – в нашем случае список значений 0 и 1 – и для каждого атрибута использует значение данного аргумента как вероятность инвертирования (применения логического оператора НЕ) значения атрибута. Это значение не зависит от вероятности мутации, которая задается константой P_MUTATION, – мы определили ее выше, но пока не использовали. Вероятность мутации нужна при решении о том, вызывать ли функцию mutFlipBit для данного индивидуума в популяции:

```
toolbox.register("select", tools.selTournament, tournsize=3)
toolbox.register("mate", tools.cxOnePoint) toolbox.register("mutate",
tools.mutFlipBit, indpb=1.0/ONE_MAX_LENGTH)
```

Итак, подготовка завершена, все операторы определены, и мы готовы реализовать генетический алгоритм.

Эволюция решения

Генетический алгоритм реализован в функции main(), его шаги описаны ниже.

1. Создаем начальную популяцию оператором populationCreator, задавая размер популяции POPULATION_SIZE. Также инициализируем переменную generationCounter, которая понадобится нам позже:

```
population = toolbox.populationCreator(n=POPULATION_SIZE)
generationCounter = 0
```

2. Для вычисления приспособленности каждого индивидуума в начальной популяции воспользуемся функцией Python map(), которая применяет оператор evaluate к каждому элементу популяции. Поскольку оператор evaluate – это псевдоним функции oneMaxFitness(), получающийся итерируемый объект содержит вычисленные значения приспособленности каждого индивидуума. Затем мы преобразуем его в список кортежей:

```
fitnessValues = list(map(toolbox.evaluate, population))
```

3. Поскольку элементы списка fitnessValues взаимно однозначно соответствуют элементам популяции (представляющей собой список индивидуумов), мы можем воспользоваться функцией zip(), чтобы объединить их попарно, сопоставив каждому индивидууму его приспособленность:

```
for individual, fitnessValue in zip(population, fitnessValues):
        individual.fitness.values = fitnessValue
```

4. Далее, так как в нашем случае имеет место приспособляемость всего с одной целью, то извлекаем первое значение из каждого кортежа приспособленности для сбора статистики:

```
fitnessValues = [individual.fitness.values[0] for individual in population]
```

5. В качестве статистики мы собираем максимальное и среднее значение приспособленности в каждом поколении. Для этого нам понадобятся два списка, создадим их:

```
maxFitnessValues = []
meanFitnessValues = []
```

6. Теперь мы готовы написать главный цикл алгоритма. В самом начале цикла проверяются условия остановки. Одно из них – ограничение на количество поколений, второе – проверка на лучшее возможное решение (двоичная строка из одних единиц):

```
while max(fitnessValues) < ONE_MAX_LENGTH and generationCounter
 < MAX_GENERATIONS:
```

7. Затем обновляется счетчик поколений. Он используется в условии остановки и в последующих предложениях печати:

```
generationCounter = generationCounter + 1
```

8. Сердце алгоритма – генетические операторы, которые применяются на следующем шаге. Сначала – оператор отбора toolbox.select, который мы выше определили как турнирный отбор. Поскольку размер турнира был задан в определении оператора, сейчас нам осталось передать только популяцию и ее размер:

```
offspring = toolbox.select(population, len(population))
```

9. Далее отобранные индивидуумы, которые находятся в списке offspring, клонируются, чтобы можно было применить к ним следующие генетические операторы, не затрагивая исходную популяцию:

```
offspring = list(map(toolbox.clone, offspring))
```

** Заметим, что, несмотря на имя offspring, это пока еще клоны индивидуумов из предыдущего поколения, и к ним еще только предстоит применить оператор скрещивания для создания потомков*

10. Следующий генетический оператор – скрещивание. Ранее мы определили его в атрибуте toolbox.mate как псевдоним одноточечного скрещивания. Мы воспользуемся встроенной в Python операцией среза, чтобы объединить в пары каждый элемент списка offspring с четным индексом со следующим за ним элементом с нечетным индексом. Затем с помощью функции random() мы «подбросим монету» с вероятностью, заданной константой P_CROSSOVER, и тем самым решим, применять к паре индивидуумов скрещивание или оставить их как есть. И наконец, удалим значения приспособленности потомков, потому что они были модифицированы и старые значения уже не актуальны:

```
for child1, child2 in zip(offspring[::2], offspring[1::2]):
    if random.random() < P_CROSSOVER:
        toolbox.mate(child1, child2)
    del child1.fitness.values
    del child2.fitness.values
```

**Отметим, что функция mate принимает двух индивидуумов и модифицирует их на месте, т. е. присваивать им новые значения не нужно.*

11. Последний генетический оператор – мутация, ранее мы определили его в атрибуте toolbox.mutate как псевдоним инвертирования бита. Мы должны обойти всех потомков и применить оператор мутации с вероятностью P_MUTATION. Если индивидуум подвергся мутации, то нужно удалить значение его приспособленности (если оно существует), поскольку оно могло быть перенесено из предыдущего поколения, а после мутации уже не актуально:

```
for mutant in offspring:
        if random.random() < P_MUTATION:
                toolbox.mutate(mutant)
                del mutant.fitness.values
```

12. Те индивидуумы, к которым не применялось ни скрещивание, ни мутация, остались неизменными, поэтому их приспособленности, вычисленные в предыдущем поколении, не нужно заново пересчитывать. В остальных индивидуумах значение приспособленности будет пустым. Мы находим этих индивидуумов, проверяя свойство valid класса Fitness, после чего вычисляем новое значение приспособленности так же, как делали это ранее:

```
freshIndividuals = [ind for ind in offspring if not ind.fitness.valid]
freshFitnessValues = list(map(toolbox.evaluate, freshIndividuals))
for individual, fitnessValue in zip(freshIndividuals, freshFitnessValues):
        individual.fitness.values = fitnessValue
```

13. После того как все генетические операторы применены, нужно заменить старую популяцию новой:

```
population[:] = offspring
```

14. Прежде чем переходить к следующей итерации, учтем в статистике текущие значения приспособленности. Поскольку приспособленность представлена кортежем (из одного элемента), необходимо указать индекс [0]:

```
fitnessValues = [ind.fitness.values[0] for ind in population]
```

15. Далее мы вычисляем максимальное и среднее значения, помещаем их в накопители и печатаем сводную информацию:

```
maxFitness = max(fitnessValues)
meanFitness = sum(fitnessValues) / len(population)
maxFitnessValues.append(maxFitness)
meanFitnessValues.append(meanFitness)
print("- Поколение {}: Макс приспособ. = {}, Средняя приспособ. = {}"
        .format(generationCounter, maxFitness, meanFitness))
```

16. Дополнительно мы находим индекс (первого) лучшего индивидуума, пользуясь только что найденным значением приспособленности, и распечатываем этого индивидуума:

```
t_index = fitnessValues.index(max(fitnessValues))
print("Лучший индивидуум = ", *population[best_index], "\n")
```

17. После срабатывания условий остановки накопители статистики можно использовать для построения двух графиков с помощью библиотеки matplotlib. Во фрагменте кода ниже рисуется график изменения лучшей и средней приспособленности:

```
plt.plot(maxFitnessValues, color='red')
plt.plot(meanFitnessValues, color='green')
plt.xlabel(Поколение') plt.ylabel('Макс/средняя приспособленность')
plt.title('Зависимость максимальной и средней приспособленности от
поколения') plt.show()
```

Теперь можно испытать свой первый генетический алгоритм – запустим его и найдем решение задачи OneMax.

Выполнение программы

В результате выполнения написанной выше программы получается такая распечатка:

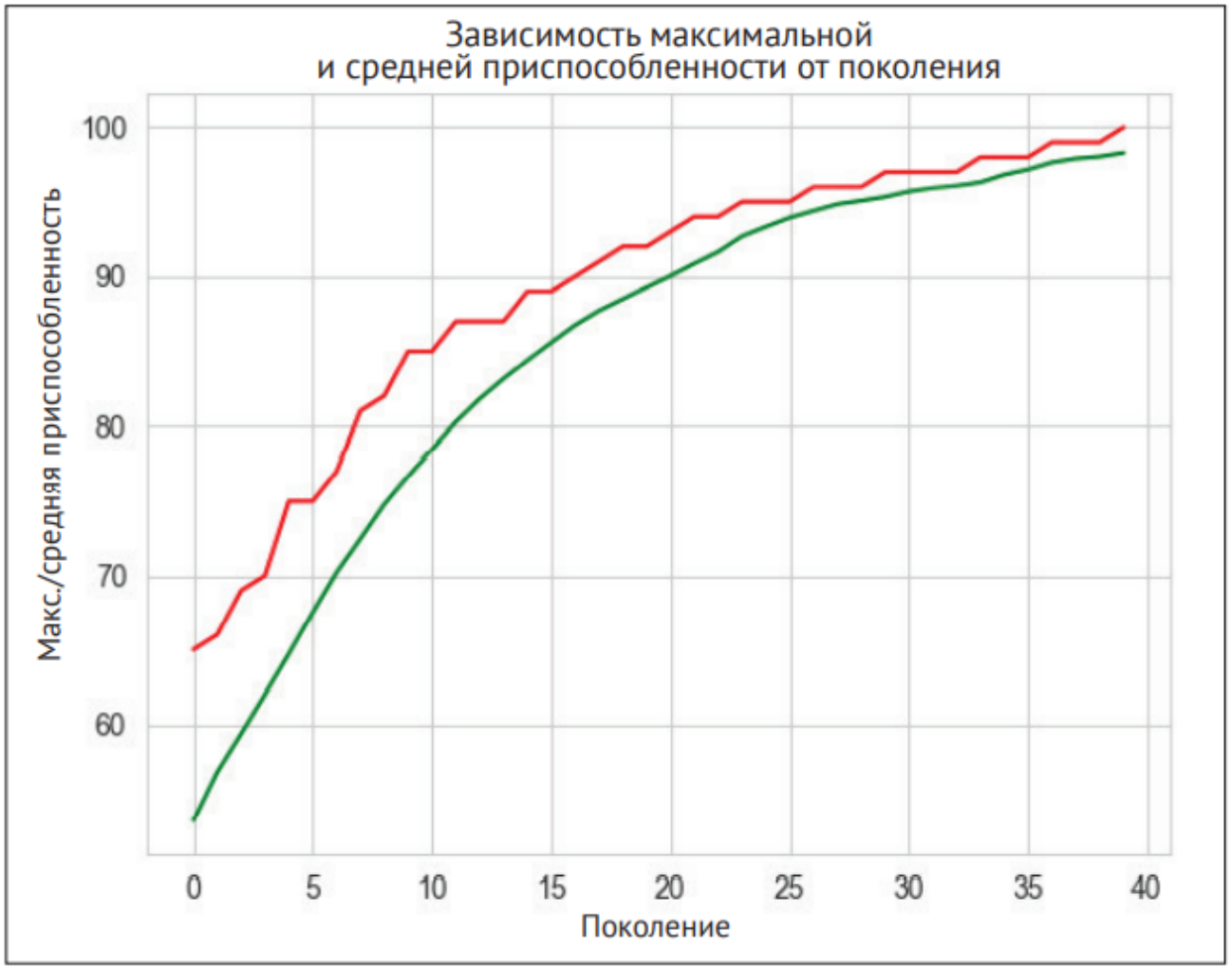
- Поколение 1: Макс приспособ. = 65.0, Средняя приспособ. = 53.575 Лучший индивидуум = 1 1 0 1 0 1 0 0 1 0 0 0 1 1 1 0 1 0 0 1 0 1 0 0 0 1 1 1 1 0 1 1 1 0 1 0 1 1 1 0 0 1 1 1 1 1 1 0 1 1 1 1 1 0 1 1 1 1 1 1 1 0 0 0 0 1 0 1 0 1 1 1 0 1 1 0 0 0 1 1 1 0 0 1 1 1 1 1 1 1 1 1 1 1 0 0

...

- Поколение 40: Макс приспособ. = 100.0, Средняя приспособ. = 98.29 Лучший индивидуум = 1

Как видим, после смены 40 поколений было найдено решение, содержащее только единицы, для которого приспособленность равна 100. После этого алгоритм остановился. Начальное значение средней приспособленности было равно примерно 53, конечное – 100.

Ниже показан график, построенный с помощью matplotlib.



Статистика программы для решения задачи OneMax

Как видно из графика, максимальная приспособленность возрастает скачкоообразно, а средняя – плавно. Итак, мы решили задачу OneMax с помощью каркаса DEAP, а теперь посмотрим, как можно сделать код более лаконичным.

Перечень задач:

- Задача о рюкзаке
- Задача коммивояжера
- Задача маршрутизации транспорта
- Задача графика дежурств (медсестер)
- Задача раскраски графа

Задача о рюкзаке

Формально в задаче о рюкзаке имеются следующие компоненты:

- набор предметов, каждый из которых имеет ценность и вес;
- рюкзак (сумка, контейнер), в который можно поместить предметы ограниченного суммарного веса.

Наша цель – отобрать группу предметов максимальной суммарной ценности, так чтобы суммарный вес не превышал емкость контейнера.

В контексте алгоритмов поиска каждый набор предметов представляет состояние, а множество всех возможных наборов считается пространством состояний. Например, в задаче о рюкзаке 0-1 с n предметами размер пространства состояний равен 2n и растет очень быстро даже при сравнительно небольших n.

В этой (классической) постановке каждый предмет либо включается ровно один раз, либо не включается вообще, поэтому иногда говорят о рюкзаке 0-1. Но возможны и другие постановки, например каждый предмет может включаться несколько раз (ограниченное или неограниченное количество) или имеется несколько рюкзаков разной емкости.

Задача о рюкзаке возникает во многих реальных процессах, где требуется принимать решения, связанные с выделением ресурсов, например: состав активов в инвестиционном портфеле, минимизация отходов при распиле, получение максимального количества баллов при решении о том, на какие вопросы отвечать в тесте с ограниченным временем.

Задача коммивояжера

Задача коммивояжера восходит к 1930 году и является одной из наиболее изученных задач оптимизации. Зачастую она используется для оценки алгоритмов оптимизации. У этой задачи много вариантов, но первоначально она ставилась на примере коммивояжера, которому требуется объехать несколько городов: Пусть дан список городов и известны расстояния между каждыми двумя городами. Найти кратчайший путь, проходящий через все города и возвращающийся в исходную точку. Легко показать, что количество возможных путей, проходящих через n городов, равно $(n - 1)!/2$

На рисунке ниже показан кратчайший путь, проходящий через 15 крупнейших городов Германии.

Кратчайший маршрут коммивояжера, проходящий через 15 крупнейших городов Германии

Источник: https://commons.wikimedia.org/wiki/File:TSP_Deutschland_3.png. Автор Kapitan Nemo. Является общественным достоянием

Поскольку в этом случае n = 15, количество возможных путей равно 14!/2 = 43 589 145 600 – пугающее число. В контексте алгоритмов поиска каждый путь (или частичный путь), проходящий через города, представляет состояние, а множество всех возможных путей рассматривается как пространство состояний. У каждого пути имеется стоимость – его длина, – и мы ищем путь минимальной стоимости. Как уже отмечалось, пространство состояний очень велико даже при сравнительно небольшом количестве городов, поэтому рассмотреть каждый путь не представляется возможным. И хотя проложить какой-то путь через все города сравнительно просто, найти оптимальный путь очень трудно.

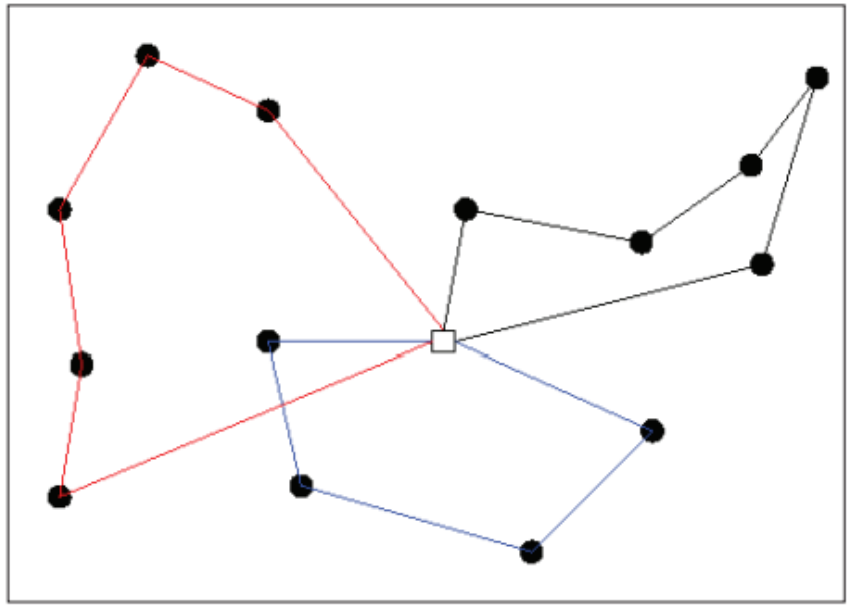
Задача о маршрутизации транспорта

Задача о маршрутизации транспорта – обобщения задачи коммивояжера. В простейшем случае задача состоит из трех компонентов:

- список подлежащих посещению адресов;
- количество автомобилей;
- местоположение гаража, который является начальной и конечной точкой каждого маршрута.

У этой задачи много вариантов, например: наличие нескольких гаражей, доставка в оговоренный срок, разные типы транспорта (скажем, разной вместимости или с разной топливной эффективностью) и т. д. Наша цель – минимизировать стоимость, которую тоже можно определить по-разному, например: время доставки всех посылок, затраты на топливо или расхождение во времени разъезда всех автомобилей.

Ниже приведен пример задачи о маршрутизации транспорта с тремя автомобилями. Города обозначены черными кружками, гараж – белым квадратиком, а маршруты трех автомобилей – разными цветами.



Пример задачи о маршрутизации транспорта с тремя автомобилями

Источник: https://commons.wikimedia.org/wiki/File:Figure_illustrating_the_vehicle_routing_problem.png. Автор PierreSelim. Является общественным достоянием

Целью является оптимизация времени доставки всех посылок. Поскольку все автомобили разъезжают одновременно, этот показатель определяется для автомобиля с самым длинным маршрутом. Поэтому можно сказать, что цель – минимизация длины самого длинного маршрута всех автомобилей доставки. Так, при наличии трех автомобилей каждое решение состоит из трех маршрутов. Мы обсчитываем все три, но для вычисления приспособленности оставляем только самый длинный – чем длиннее этот маршрут, тем хуже оценка. Поэтому неявно ставится задача сократить протяженность всех трех маршрутов, а также по возможности уравнять их длины. Поскольку задачи коммивояжера и маршрутизации транспорта похожи, мы можем воспользоваться ранее написанным кодом. Для этого представим данные в задаче о маршрутизации следующим образом:

- экземпляр задачи коммивояжера, а именно список городов с координатами (или попарные расстояния между ними);
- местоположение гаража, совпадающее с одним из городов и представленное индексом этого город
- количество автомобилей.

Задача о составлении графика дежурств медсестер

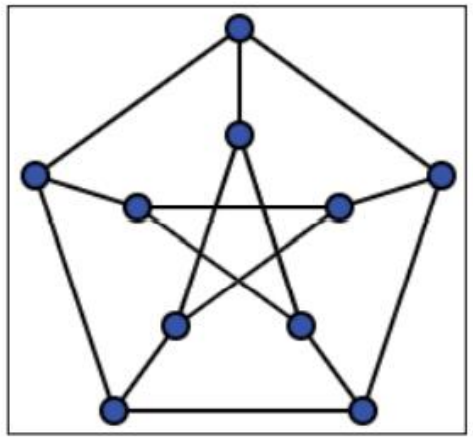
Требуется составить недельный график дежурств медсестер в отделении больницы. В сутках три смены – утренняя, дневная и вечерняя, и в каждой смене должно быть не менее одной из восьми работающих в отделении медсестер. Если вам кажется, что это просто, примите во внимание перечень действующих в больнице правил:

- сестре не разрешается работать две смены подряд;
- сестре не разрешается работать более пяти смен в неделю;
- количество сестер в смене должно удовлетворять дополнительным ограничениям: – в утренней смене:
  – 2–3 сестры;
  – в дневной смене: 2–4 сестры;
  – в ночной смене: 1–2 сестры.

Кроме того, у каждой сестры могут быть пожелания. Например, одна сестра предпочитает только утренние смены, другая не хочет работать днем и т. д. Эта задача о составлении графика дежурств медсестер (СГДМ) имеет много вариантов. Например, у сестер могут быть различные специальности, можно разрешить сверхурочную работу, даже сами смены могут быть разными – скажем, 8- и 12-часовыми.

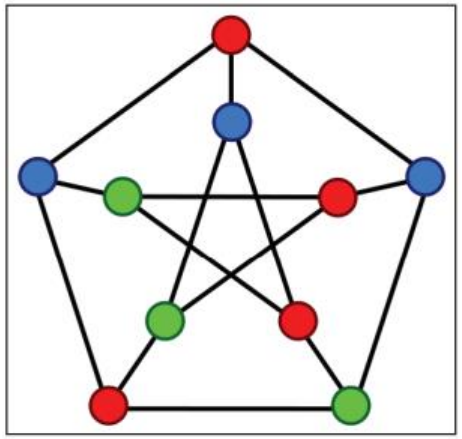
Задача о раскраске графа

Задача о раскраске графа заключается в том, чтобы назначить каждой вершине цвет таким образом, чтобы никакие две пары смежных (соединенных ребром) вершин не были выкрашены в один цвет. Такая раскраска называется правильной



Граф Петерсена

Источник: https://commons.wikimedia.org/wiki/File:Petersen1_tiny.svg.
Автор: Leshabirukov. Публикуется по лицензии Creative Commons CC BY-SA 3.0: https://creativecommons.org/licenses/by-sa/3.0/deed.en



Правильная раскраска графа Петерсена

Источник: https://en.wikipedia.org/wiki/File:Petersen_graph_3-coloring.svg.
Является общественным достоянием

На раскраску часто налагается дополнительное требование – использовать минимально возможное число цветов. Например, граф Петерсена можно раскрасить в три цвета (см. рисунок выше). Но в два цвета его раскрасить невозможно. В теории графов говорят, что хроматическое число этого графа равно трем

Почему раскраска вершин графа так интересна?

Многие реальные задачи можно сформулировать таким образом, что для их решения нужно раскрасить некоторый граф. Например, составление расписания занятий для студентов или сменного графика для рабочих можно рассматривать как граф, в котором смежные вершины представляют занятия или смены, приводящие к конфликту. Конфликтом могут быть лекции, проходящие в одно и то же время, или несколько смен подряд (знакомо звучит?). Поэтому назначение одного лица на обе лекции (или в обе смены) сделает расписание недопустимым. Если каждому лицу сопоставить цвет, то раскрашивание смежных вершин в разные цвета устранит все конфликты. Задачу об N ферзях можно сформулировать как задачу о раскраске графа: вершинами графа являются поля шахматной доски, а вершины, находящиеся на одной горизонтали, вертикали или диагонали, соединены ребром. Можно привести и другие примеры: распределение частот между радиостанциями, планирование избыточности электрических сетей, время включения и выключения светофоров и даже решение головоломки судоку.