

# ComputeGraph Homework

---

## Trivia

В этом задании вам надо будет реализовать библиотеку для удобного запуска вычислений над таблицами.

Таблица - это последовательность dict-like объектов (собственно словарей или, например, json-строк задающих словари), где каждый словарь — это строка таблицы, а ключ словаря — колонка таблицы (индекс в последовательности + ключ в словаре задают ячейку). Для простоты будем считать, что в таблице в каждой строчке все ячейки определены, то есть в каждом словаре одинаковый набор ключей.

Над таблицами мы хотим запускать вычисления, задавая их с помощью *Вычислительных графов*. Ваша библиотека должна позволять задавать цепочку преобразований над табличками так, чтобы выполнение этих преобразований производилось отдельно от их задания. То есть хочется как бы описать "рецепт" многоступенчатой обработки нескольких таблиц, и уметь применять его к разным таблицам.

## Зачем могут понадобиться вычислительные графы?

Благодаря тому, что мы отделяем задание последовательности операций от их выполнения, мы можем создавать и выполнять вычислительные графы в разных средах.

Создав один раз граф вычислений в скрипте на питоне, его можно сохранить и загрузить в видеокарту вашего компьютера и выполнить там - [так работает библиотека для обучения нейросетей tensorflow от Google](#).

Или же можно передать граф по сети сразу на большое количество машин и запустить на них параллельно - так работает клиент к системе распределенных вычислений Spark.

В общем, эта идиома в реальной жизни встречается довольно часто, и, несмотря на то, что мы будем выполнять графы не на видеокартах или больших кластерах, а всего лишь на ваших ноутбуках, там же где они были созданы, - понять как писать код в таком стиле полезно.

## Интерфейс графа вычислений

Граф вычислений состоит из входов и операций над ними. В момент создания графа мы **не производим никаких чтений и вычислений(!)**.

При вычислении графа на входы будут подаваться открытые файлы с таблицами в виде последовательностей json-словарей, задающих строки. Нужно придумать способ задавать входы при создании графа так, чтобы при запуске можно было удобно передать файлы в соответствующие входы.

Над входами мы запускаем операции (о типах операций ниже). В общем случае операции вызываются последовательно, но есть операции (join), которые на вход принимают другие графы вычислений; за счет этого полный граф вычислений может быть нелинеен.

После задания граф нужно уметь запускать, передав ему все нужные входы и файл для записи результата.

Пример возможного интерфейса, не лишенный проблем (подумайте как сделать лучше и удобнее):

```
import mrop # your lib
best_graph = mrop.ComputeGraph(source='main_input')
best_graph.add_operation(mrop.Map(mapper=mapper_func))
best_graph.add_operation(mrop.Sort('column2', 'column3'))
best_graph.add_operation(mrop.Reduce(reducer=reducer_func, key=('column2',
'column3')))

best_graph.run(
    main_input=open('source.txt'),
    save_result=open('result.txt', 'w'),
    verbose=True
)
```

## Операции

Казалось бы, операции над таблицами можно объявлять просто, как функции, принимающие таблицу и возвращающие новую. Но мы поступим иначе: наши операции будут работать со строками таблицы, а не с таблицей целиком. Так их будет проще писать и применять - можно будет избегать лишних копирований и много чего еще.

Для простоты ограничим спектр возможных операций над таблицами этими 5-ю:

1. **Map** — операция, которая вызывает переданный генератор (называемый мэппером) от каждой из строк таблицы. Значения, выданные генератором, образуют таблицу-результат. (Подходит для элементарных операций над строками - фильтраций, преобразований типов, элементарных операций над полями таблицы etc).

Мэп в нашем задании не тождественен функции map из python, которая реализует соответствие 1-к-1 (наша по каждой строке может вернуть любое нетрицательное число строк).

Мэппер может быть, например, таким:

```
def tokenizer_mapper(r):
    """
    splits rows with 'text' field into set of rows with 'token' field
    (one for every occurrence of every word in text)
    """

    tokens = r['text'].split()

    for token in tokens:
        yield {
            'doc_id' : r['doc_id'],
            'word' : token,
        }
```

Этот мэппер работает с табличками, в которых есть колонки text и doc\_id, превращая каждую их строку в набор строк - по одной на каждое слово в тексте

```
def minlen_filter_mapper(r):
    """
    filters out values with len(data) < 10
    """
    if len(r['data']) > 10:
        yield r
```

А этот мэппер фильтрует исходную таблицу и оставляет только строки, в которых поле Data длиннее 10 символов

2. **Sort** — сортирует таблицу по какому-то набору колонок лексикографически.
3. **Fold** — "сворачивает" таблицу в единую строку с помощью бинарной ассоциативной операции.

Для запуска Fold необходимы два аргумента - комбинирующая функция folder и начальное состояние. Fold должен последовательно вызывать переданный ему фолдер на паре (состояние, новая строка) и возвращать измененное состояние. Результатом операции будет это самое состояние после того, как к нему будут применены все строки. [Описание этой операции на вики](#).

Пример фолдера, считающего сумму каких-то колонок таблицы

```
def sum_columns_folder(state, record):
    for column in state:
        state[column] += record[column]
    return state
```

4. **Reduce** — операция, похожая на map, но вызываемая не для одной строки таблицы, а для всех строк с одинаковым значением ключа (где ключ - значение какого-то подмножества колонок таблицы). Эти строки передаются в редьюсер как итерируемая последовательность. Для эффективной работы редьюсера (за  $O(n)$ ) таблица, подаваемая ему на вход, должна быть отсортирована по колонкам, на которых он запускается. Это надо проверять.

Пример редьюсера:

```
def term_frequency_reducer(records):  
    '''  
        calculates term frequency for every word in doc_id  
    '''  
  
    word_count = Counter()  
  
    for r in records:  
        word_count[r['word']] += 1  
  
    total = sum(word_count.values)  
    for w, count in word_count.items():  
        yield {  
            'doc_id' : r['doc_id'],  
            'word' : w,  
            'tf' : count / total  
        }
```

Редьюсер, как мы договорились выше, принимает итерируемую последовательность строк с одним ключом — в данном случае с одним doc\_id (документом). Для каждого слова этого документа, редьюсер считает его частоту. Результатом является табличка с тремя колонками (документ, слово, частота).

5. **Join** — операция объединяющая информацию из двух таблиц в одну по ключу. Строки новой таблицы будут созданы из строк двух таблиц участвовавших в джойне. Для join надо реализовать несколько стратегий — (outer, left, right, inner). Если не знаете что такое join двух таблиц — читайте статью [Join \(SQL\) на википедии](#) (понятно что не нужно реализовывать синтаксис SQL, только концептуальную идею). Выглядеть задание этой операции должно как-то так.

```
Join(on=other_graph, key=('key1', 'key2'), strategy='left')
```

Джойним мы, по факту, один граф вычислений с результатом другого.

Других операций у вас быть не должно.

## Зависимости между графами

Обратите внимание, что функция `join` принимает на вход другой вычислительный граф и `join` надо будет делать с его результатом. Соответственно, при вычислении надо будет вычислить и этот граф тоже - а для этого предоставить ему все нужные входы. При этом возможны ромбовидные зависимости — например граф A зависит от B и C, и B тоже зависит от C. Хорошо бы сделать так, чтобы C при этом вычислялся только один раз.

## Собственно, требования

1. Спроектировать и закодировать библиотеку для создания и использования вычислительных графов. Подумать о том, чтобы у неё был максимально удобный и красивый интерфейс. Библиотеку надо разместить в приватном репозитории на [gitlab.com](https://gitlab.com) и дать к ней доступ для [[@ph.sinitsyn](#) @Gezort и @jhilary]. [Ридми гитлаба, читать если не знаете как заливать в него ваш проект.](#)
2. Для библиотеки написать юнит-тесты, используя библиотечку `pytest`, и тоже залить их в репозиторий (подумав об удобной структуре каталогов - не надо валить всё в корень)
3. Написать описание своей библиотеки в файле `README` и снабдить весь код `docstring`-ами (правильный формат `dostring`-ов описан в PEP-0257).
4. Используя вашу библиотеку решить четыре задачи, приведенные ниже (при этом задача должна целиком решаться в модели вычислительного графа - от чтения входа до записи результата). Код для решения задач так же положите в репозиторий с библиотекой, оформив его, как примеры её использования (например в папку `examples`, и, конечно же, надо сделать для неё своё `README`).
5. Код основного модуля библиотеки залить на `codereview` в энитаск, в комментариях указать ссылку на репозиторий со всей библиотекой. Оценка выставляется при успешном прохождении этого `codereview`.
6. Для удобства отладки реализуйте в методе `run` вашего графа параметр `verbose`, который будет выводить на экран выполняемые при запуске графа операции.
7. Частичное решение - тоже решение, но постарайтесь справиться с заданием целиком. Оно не такое трудное, как может показаться.

## Задачи для решения с помощью вашей библиотеки

---

### Word Count

Задача для разминки.

В этой задаче вам дана коллекция документов в формате `{ 'doc_id': 'name', 'text': ... }`. То есть таблица с колонками заголовков - текст. Требуется для каждого из слов, встречающихся в текстах, посчитать количество вхождений во все тексты в сумме.

Файлы с данными для этой и двух следующих задач — `text_corpus.txt` (лежит на вики)

### Инвертированный индекс на `tf-idf`

Работать будем с тем же датасетом, что и в предыдущей задаче. Для этой коллекции построим *инвертированный индекс* — структуру данных, которая для каждого слова хранит список документов, в котором оно встречается, отсортированный в порядке *релевантности*. Релевантность будем считать по метрике [tf-idf](#).

Для каждой пары (слово, документ) tf-idf зададим так:

$$TFIDF(word_i, doc_i) = (frequency\ of\ word_i\ in\ doc_i) \cdot \log\left(\frac{|total\ number\ of\ docs|}{|docs\ where\ word_i\ is\ present|}\right)$$

Результат должен выглядеть так: `{ 'term' : 'word', 'index' : [(doc_id_1, tf_idf_1)...]}`

Для каждого слова надо посчитать топ-3 документов по tf-idf.

Когда будете делить текст на слова, не забудьте убрать пунктуацию.

(Эта задача разобрана ниже)

## Топ слов с наибольшей взаимной информацией

Задача, обратная предыдущей: на том же датасете надо для каждого документа посчитать топ-10 слов наиболее характерных для него. Ранжировать слова будем по метрике [Pointwise mutual information](#). Более формально задача ставится так: для каждого текста надо найти топ-10 слов, каждое из которых длиннее четырех символов и встречаются в каждом из документов не менее двух раз; топ надо выбирать по величине

$$pmi(word_i, doc_i) = \log\left(\frac{frequency\ of\ word_i\ in\ doc_i}{frequency\ of\ word_i\ in\ all\ documents\ combined}\right)$$

## Средняя скорость движения по городу от часа и дня недели

В этой задаче вам надо работать с информацией о движении людей на машинах по какому-то подмножеству улиц города Москвы. Улицы города заданы как граф, а информация о передвижении задана как таблица, в каждой строке которой данные вида

```
{ 'edge_id':  
'624', 'enter_time': '20170912T123410.1794', 'leave_time': '20170912T123412.68' }
```

где edge\_id — идентификатор ребра графа дорог (то есть просто участка какой-то улицы), enter\_time и leave\_time — соответственно время въезда и выезда на это ребро (время в utc)

Также вам дана вспомогательная таблица вида

```
{ 'edge_id': '313', 'length': 121, 'start': (37.31245, 51.256734), 'end': (37.31245,  
51.256734) }
```

где length - длина в метрах, start и end — координаты начала и конца ребра, заданные в формате ('lon', 'lat'). Быть может, не для всех рёбер графа есть вся метаданная информация.

Пользуясь этой информацией вам надо построить таблицу со средней скоростью движения по городу в км/ч в зависимости от часа и дня недели.

```
{ 'weekday' : 'Mon', 'hour' : 4, 'speed' : 44.812 }
```

Для проверки постройте график по этой таблице, он должен выглядеть предсказуемо.

Файлы для этой задачи — `travel_times.txt` и `graph_data.txt`

---

## Разбор задачи про tf-idf

Решим задачу обратного индекса на tf-idf.

На входе нам дана таблица с колонками ("doc\_id", "text"). Т.е. просто список документов с названиями.

На выходе для каждого слова из исходных документов хотим получить последовательность doc\_id в порядке убывания tf-idf для этого слова в этом документе.

Решим задачу в несколько этапов:

1. Создадим граф *split\_word*, который пропустит вход нашей исходной таблички через приведенный выше (в описании мэпперов) мэппер, который разделит каждую строку с текстом на много строк - по одной для каждого слова в тексте.

```
split_word := input('docs') -> map(split_words)
```

2. Создадим граф *count\_docs*, который посчитает количество документов в таблице (оно нужно нам для формулы idf). Он будет состоять из единственной функции fold, которая будет считать количество строчек в переданной ей таблице. Результатом этого графа будет таблица с единственной строкой `{ 'docs_count' : n }`

```
count_docs := input('docs') -> fold(count_rows)
```

3. Создадим граф *count\_idf*, считающий idf каждого слова. Входом для этого графа будет выход графа *split\_word*.

Для начала запустим на этой таблице reduce по ключу ('doc\_id', 'word') и для каждого ключа оставим только одно его вхождение (ведь нас интересует количество документов в которых встретилось слово, без разницы сколько раз).

Далее приджойним count\_docs к каждой строчке нашей таблицы (он нужен для того, чтобы посчитать idf для каждого слова). Для этого будем пользоваться outer join.

Результат этой операции будем сортировать и редьюсить уже по словам (т. е. по 'word') - в нем будем считать количество документов, в которых это слово встретилось. Результатом графа является таблица где для каждого слова посчитан знаменатель формулы tf-idf.

```
count_idf := input(split_words) -> sort('doc_id', 'word') ->
reduce(unique, keys=('doc_id', 'word')) -> join(count_docs,
type='outer') -> sort('word') -> reduce(calc_idf, keys=('word'))
```

4. Создадим третий граф, *calc\_index*. Входом для него так же, как для предыдущего графа, будет результат графа *split\_word* - но в этот раз мы его будем редьюсить по документам (по *doc\_id*) и внутри редьюсера будем считать частоту каждого слова, то есть числитель нужной нам формулы (Этот редьюсер приведет в разборе про редьюсеры).

После этого нужно сделать join с результатом графа *count\_idf*. Теперь у нас для каждой пары документ-слово есть и *tf*, и *idf* — всё что нужно для победы. Последним будет *reduce* по 'word', считающий ответ — то есть *tf-idf* для каждой пары слово-документ - и оставляющий для каждого слова *top-3* документа по *tf-idf*.

```
calc_index := input(split_word) -> sort('document_id') -> reduce(tf,
'document_id') -> join(count_idf, keys='word', 'type='left') ->
reduce(invert_index, key = 'word')
```

Код для графа *calc\_index* может выглядеть как-то так:

```
calc_index = mrop.ComputeGraph() \
    .input(split_words) \
    .sort('document_id') \
    .reduce(tf, keys='document_id') \
    .join(idf, keys='word', type='left') \
    .sort('word') \
    .reduce(invert_index, keys='word')

index = calc_index.compute(docs='docs.txt', save=open('tf-idf.txt'))
```

Такой интерфейс определения графа, как и приведенный выше, тоже не лишен проблем. Постарайтесь придумать лучше.

Теперь для того, чтобы построить инвертированный индекс, достаточно вызвать метод *compute* у графа *calc\_index*, передав на вход все входы, нужные ему и графам, от которых он зависит, то есть в данном случае просто файл *doc* для графа *\*split\_word\** (обратите внимание, что самому *calc\_index* *doc* не нужен, и он должен как-то передать его графам, от которых он зависит).

## Hints:

1. Удобно (и просто) большую часть операций применять в генераторах, чтобы минимизировать копирование таблицы в графах.
2. Для того, чтобы посчитать в каком порядке выполнять все графы от которых зависит данный потребуется [Топологическая сортировка](#). Это не сложный алгоритм, разберитесь.
3. Проще всего задачу решать в таком порядке:
  - Спроектировать интерфейс библиотеки (то есть, просто имена классов и названия методов)
  - Воспользовавшись библиотекой как готовой решить одну из тестовых задач
  - После этого дописать реализацию использованных методов в свою библиотеку,



и заставить задачу работать.

- Повторить с более сложными задачами.