Большие данные

613x-010402D x={1,2,3} *осень 2025*

Лабораторная работа №1

Введение в модель MapReduce

Сергей Борисович Попов
sepo@ssau.ru

Материалы лекций:
https://1drv.ms/f/c/5ed33c8b23e26391/EpMzlOhQgt5Oq-cqHF_ChjwB0NP5AovVgqYTfBokEY1zhA

Цель: реализация алгоритмов в парадигме MapReduce на модели фреймворка MapReduce, реализованной на Python

Алгоритмы:

1. Алгоритм вычисления TF-IDF для корпуса документов
2. Алгоритм поиска кратчайшего пути на графе с использованием параллельного поиска в ширину (Breadth-First Search)

Исходные данные:

1. Для алгоритма вычисления TF-IDF корпус документов: строки, содержащие аннотации статей.
2. Список смежности графа

MapReduce

- Программист определяет две основные функции:

  map (k1, v1) → list(k2, v2)

  reduce (k2, list(v2*) → list(k3, v3)

  – Все значения с одинаковым ключом отправляются на один и тот же reducer

- … и опционально:

  partition (k2, number of partitions) → partition for k2

  – Часто просто хеш от key, напр., *hash(k2) mod n*

  – Разделяет множество ключей для параллельных операций reduce

  combine (k2, v2) → list(k2, v2')

  – Мини-reducers которые выполняются после завершения фазы map

  – Используется в качестве оптимизации для снижения сетевого трафика на reduce

Модель фреймворка MapReduce, реализованная на Python

[ ]:
```
def flatten(nested_iterable):
  for iterable in nested_iterable:
    for element in iterable:
      yield element

def groupbykey(iterable):
  t = {}
  for (k2, v2) in iterable:
    t[k2] = t.get(k2, []) + [v2]
  return t.items()

def MapReduce(RECORDREADER, MAP, REDUCE):
  return flatten(map(lambda x: REDUCE(*x), groupbykey(flatten(map(lambda x: MAP(*x), RECORDREADER())))))
```

Пользователь для решения своей задачи реализует RECORDREADER, MAP, REDUCE.

Пример использования модели фреймворка MapReduce для реализации алгоритма Word Count

WordCount

[17]:
```
from typing import Iterator

d1 = """it is what it is
it is what it is
it is what it is"""
d2 = """what is it
what is it"""
d3 = """it is a banana"""
documents = [d1, d2, d3]

def RECORDREADER():
  for (docid, document) in enumerate(documents):
    for (lineid, line) in enumerate(document.split('\n')):
      yield ("{}:{}".format(docid,lineid), line)

def MAP(docId:str, line:str):
  for word in line.split(" "):
    yield (word, 1)

def REDUCE(word:str, counts:Iterator[int]):
  sum = 0
  for c in counts:
    sum += c
  yield (word, sum)

input = list(RECORDREADER())
output = MapReduce(RECORDREADER, MAP, REDUCE)
output = list(output)
output
```

```
[('0:0', 'it is what it is'),
 ('0:1', 'it is what it is'),
 ('0:2', 'it is what it is'),
 ('1:0', 'what is it'),
 ('1:1', 'what is it'),
 ('2:0', 'it is a banana')]
```

[17]:
```
[('it', 9), ('is', 9), ('what', 5), ('a', 1), ('banana', 1)]
```

Внимание!

В лабораторной работе функции RECORDREADER, MAP, REDUCE реализуются строго в рамках модели фреймворка MapReduce, то есть

- функция MAP должна иметь два входных параметра: ключ и значение, на выходе (при каждом вызове) функция MAP должна генерировать пару «ключ – значение»;
- функция REDUCE на входе имеет два параметра: ключ и список значений, на выходе генерируются пары «ключ – значение»;
- функция RECORDREADER может иметь на входе необязательный параметр исходного набора данных, на выходе генерируется список пар «ключ – значение» либо с использованием оператора *return*, либо оператора *yield*.

```
map     (k1, v1) -> (k2,v2)*
reduce (k2, v2*) -> (k3,v3)*
```

Реализация многоэтапного алгоритма предполагает, что результат работы первого задания (список пар «ключ – значение») без каких-либо изменений является входным набором данных для второго задания и т.д.

Алгоритм 1 TF-IDF

- Term Frequency – Inverse Document Frequency
  - Используется при работе с текстом
  - В Information Retrieval

- **TF** (*term frequency* — частота слова) — отношение числа вхождения некоторого слова к общему количеству слов документа.
  - Таким образом, оценивается важность слова в пределах отдельного документа.

$$\mathrm{tf}(t,d) = \frac{n_i}{\sum_k n_k}$$

где n_i есть число вхождений слова в документ, а в знаменателе — общее число слов в данном документе.

- **IDF** (*inverse document frequency* — обратная частота документа) — инверсия частоты, с которой некоторое слово встречается в документах коллекции.

$$\mathrm{idf}(t,D) = \log \frac{|D|}{|(d_i \supset t_i)|}$$

Где:
  - $|D|$ — количество документов в корпусе;
  - $|(d_i \supset t_i)|$ — кол-во документов, в которых встречается t_i (когда $n_i \neq 0$).

$$\mathrm{tfidf}(t,d,D) = \mathrm{tf}(t,d) \times \mathrm{idf}(t,D)$$

Вычисление параметра TF-IDF

Что нужно будет вычислить

- Сколько раз слово *T* встречается в данном документе (*tn*)
- Сколько слов в документе (*cn*)
- Сколько документов, в котором встречается данное слово *T* (*n*)
- Общее число документов (*N*) (предполагается, что это число известно и не вычисляется)

TF-IDF

- **Job 1**: Частота слова в документе
- *Mapper*
  - Input: *(docname, contents)*
    - Для каждого слова в документе надо сгенерить пару *(word, docname)*
  - Output: *((word, docname), 1)*
- *Reducer*
  - Суммирует число слов в документе
  - Outputs: *((word, docname), tf)*
- *Combiner* такой же как и *Reducer*

Здесь где-то надо подсчитать число всех слов в документе *cn* для вычисления *tf = tn / cn*

TF-IDF

- **Job 2**: Кол-во документов для слова
- *Mapper*
  - Input: *((word, docname), tf)*
  - Output: *(word, (docname, tf, 1))*
- *Reducer*
  - Суммирует единицы чтобы посчитать n
  - Output: *((word, docname), (tf,n))*

TF-IDF

- **Job 3**: Расчет TF-IDF
- *Mapper*
  - Input: *((word, docname), (tf,n))*
    - Подразумевается, что N известно (его легко подсчитать)
  - Output: *((word, docname), (TF*IDF))*
- *Reducer*
  - Не требуется

Алгоритм 1: Вычисление TF-IDF

Реализуется в три этапа:
Этап 1: Частота слова в документе
Этап 2: Количество документов, в которых встречается слово
Этап 3: Расчёт TF-IDF

Реализация включает:
1. Формирование набора исходных данных: проиндексированные строки, содержащие текст аннотаций.
2. Код функций RECORDREADER_1, MAP_1, REDUCE_1 для этапа 1.
3. Код функций RECORDREADER_2, MAP_2, REDUCE_2 для этапа 2.
4. Код функций RECORDREADER_3, MAP_3, REDUCE_3 для этапа 3.
5. Последовательный вызов модельной функции Mapreduce для всех трёх этапов.
6. Ввод на печать результата отдельно для каждого документа в виде первых пяти слов, упорядоченных по убыванию их значений TF-IDF (для каждого документа вывести на печать первые пять слов с максимальным TF-IDF).

Алгоритм 2 Обработка графов в парадигме **MapReduce**

Граф как структура данных:

- *G = (V,E)*, где
  - *V* представляет собой множество вершин (*nodes*)
  - *E* представляет собой множество ребер (*edges/links*)
  - Ребра и вершины могут содержать дополнительную информацию
- Различные типы графов
  - Направленные и ненаправленные
  - С циклами и без
- Графы есть практически везде
  - Структура компьютеров и серверов Интернет
  - Сайты/страницы и ссылки в Web
  - Социальные сети
  - Структура дорог/жд/метро и т.д.

Графы и MapReduce

- Большой класс алгоритмов на графах включает
  - Выполнение вычислений на каждой ноде
  - Обход графа
- Ключевые вопросы
  - Как представить граф на MapReduce?
  - Как обходить граф на MapReduce?

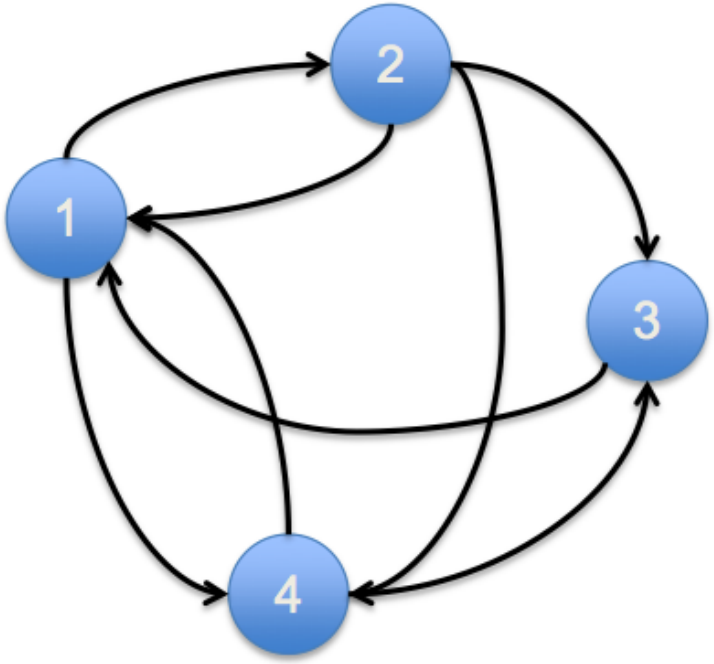
Представление графов

- G = (V, E)
  - Матрица смежности
  - Списки смежности

Матрица смежности

- Граф представляется как матрица *M* размером *n x n*
  - *n* = |*V*|
  - *Mij* = 1 означает наличие ребра между *i* и *j*

	1	**2**	**3**	**4**
1	0	1	0	1
2	1	0	1	1
3	1	0	0	0
4	1	0	1	0



Матрица смежности

- Плюсы
  - Удобство математических вычислений
  - Перемещение по строкам и колонкам соот-ет переходу по входящим и исходящим ссылкам
- Минусы
  - Матрица разреженная, множество лишних нулей
  - Расходуется много лишнего места

Списки смежности

- Берем матрицу смежности и убираем все нули

	1	**2**	**3**	**4**
1	0	1	0	1
2	1	0	1	1
3	1	0	0	0
4	1	0	1	0

1: 2, 4
2: 1, 3, 4
3: 1
4: 1, 3

Списки смежности

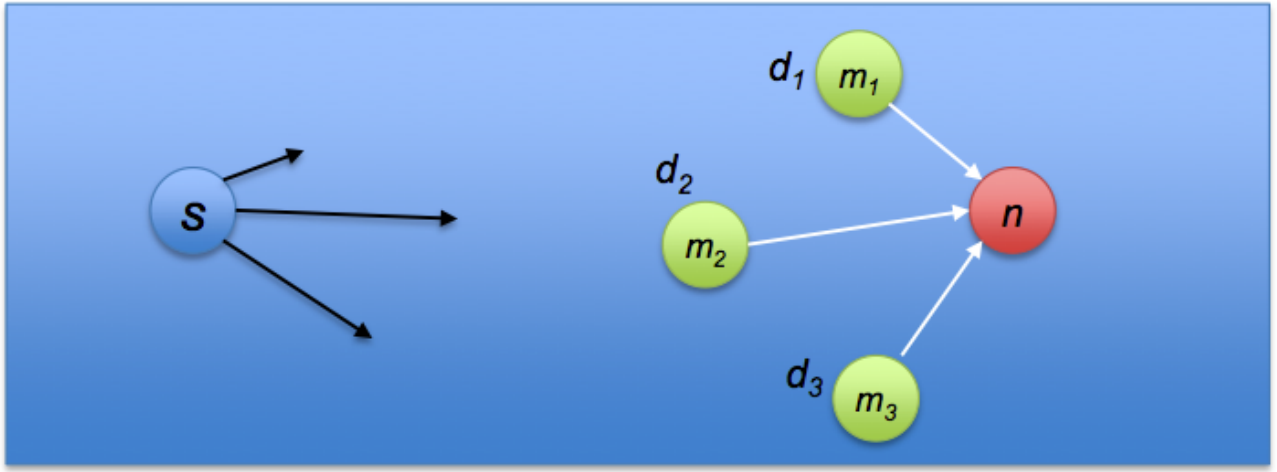
- Плюсы
  - Намного более компактная реализация
  - Легко найти все исходящие ссылки для ноды
- Минусы
  - Намного более сложнее подсчитать входящие ссылки

Поиск кратчайшего пути

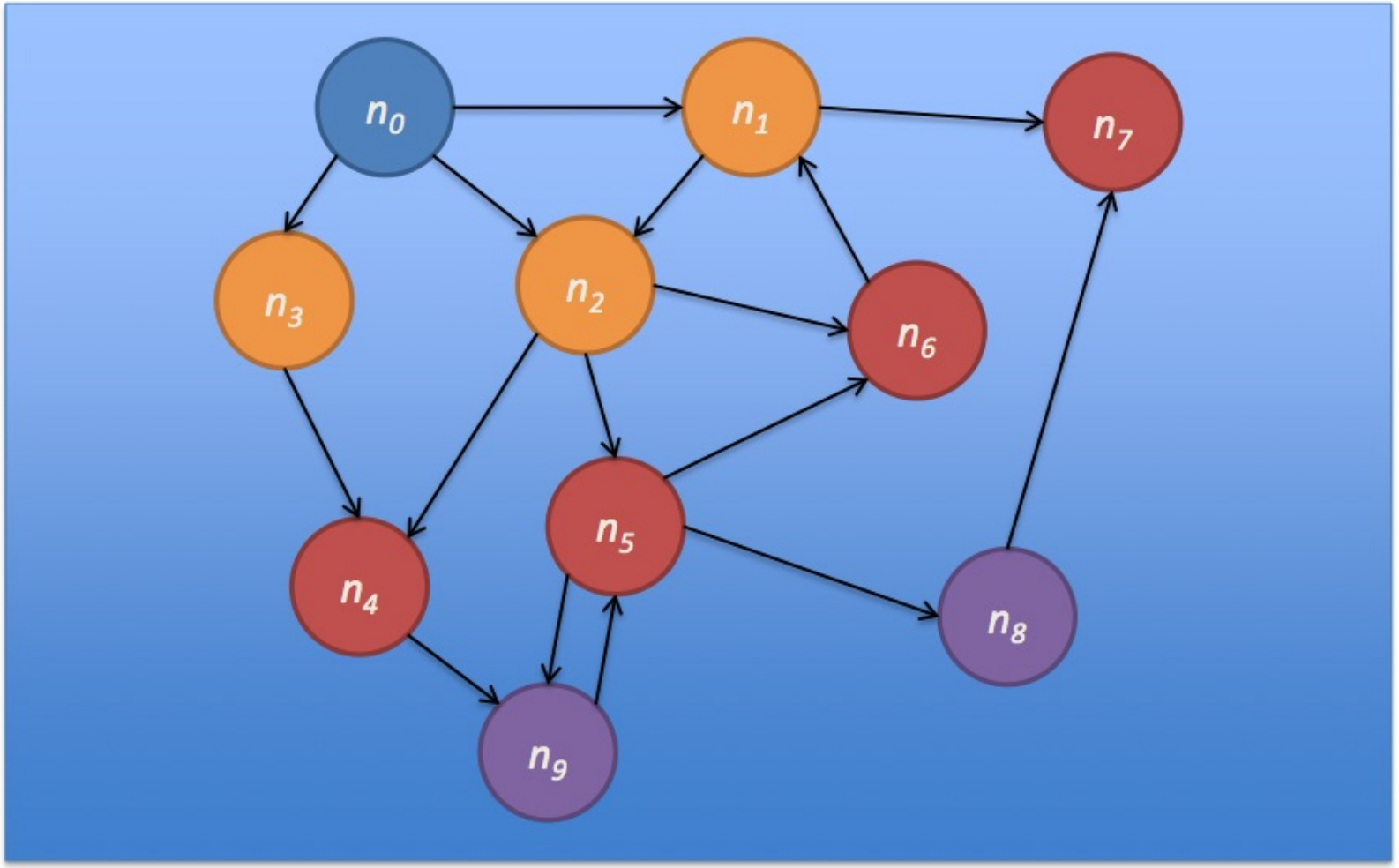
- Задача
  - Найти кратчайший путь от исходной вершины до заданной (или несколько заданных)
    - Также, кратчайший может означать с наименьшим общем весом всех ребер
  - Single-Source Shortest Path
- Алгоритм Дейкстры
- MapReduce: параллельный поиск в ширину (Breadth-First Search)

Поиск кратчайшего пути

- Рассмотрим простой случай, когда вес всех ребер одинаков
- Решение проблемы можно решить по индукции
- Интуитивно:
  - Определим: в вершину *b* можно попасть из вершины *a* только если *b* есть в списке вершин *a*
    DistanceTo(s) = 0
  - Для всех вершин *p*, достижимых из s
    DistanceTo(p) = 1
  - Для всех вершин n, достижимых из других множеств *M*
    DistanceTo(n) = 1 + min(DistanceTo(m), m ∈ M)



Параллельный BFS



BFS: алгоритм

- Представление данных:
  - Key: вершина *n*
  - Value: *d* (расстояние от начала), adjacency list (вершины, доступные из *n*)
  - Инициализация: для всех вершин, кроме начальной, $d = \infty$
- Mapper:
  - $\forall m \in$ adjacency list: emit $(m, d + 1)$
- Sort/Shuffle
  - Сгруппировать расстояния по достижимым вершинам
- Reducer:
  - Выбрать путь с минимальным расстоянием для каждой достижимой вершины
  - Дополнительные проверки для отслеживания актуального пути

BFS: итерации

- Каждая итерация задачи MapReduce смещает границу продвижения по графу (*frontier*) на один “hop”
  - Последующие операции включают все больше и больше посещенных вершин, т.к. граница (*frontier*) расширяется
  - Множество итераций требуется для обхода всего графа
- Сохранение структуры графа
  - Проблема: что делать со списком смежных вершин (*adjacency list*)?
  - Решение: Mapper также пишет *(n, adjacency list)*

BFS: псевдокод

```
class Mapper
    method Map(nid n, node N)
        d ← N.Distance
        Emit(nid n,N)                          // Pass along graph structure
        for all nodeid m ∈N.AdjacencyList do
            Emit(nid m, d + 1)                 // Emit distances to reachable nodes

class Reducer
    method Reduce(nid m, [d1, d2, . . .])
        dmin←∞
        M ← ∅
        for all d ∈counts [d1, d2, . . .] do
            if IsNode(d) then
                M ← d                          // Recover graph structure
            else if d < dmin then  Look for shorter distance
                dmin ← d
        M.Distance← dmin                       // Update shortest distance
        Emit(nid m, node M)
```

На входе связный ориентированный граф, представленный в виде списков смежности. Расстояние до каждого узла сохраняется непосредственно рядом со списком смежности этого узла и инициализируется значением ∞ для всех узлов, кроме исходного узла.
В псевдокоде используется *n* для обозначения идентификатора узла (целое число) и *N* для обозначения соответствующей структуры данных узла (список смежности и текущее расстояние).
Алгоритм работает путем сопоставления всех узлов и выдачи пары ключ-значение для *каждого* соседа в списке смежности узла. Ключ содержит идентификатор соседнего узла, а значение равно текущему расстоянию до узла плюс один. Это говорит о том, что если мы можем достичь узла *n* на расстоянии *d*, то мы должны иметь возможность достичь всех узлов, которые соединены с *n* на расстоянии *d* + 1.
После перетасовки и сортировки редьюсеры получат ключи, соответствующие идентификаторам узлов назначения, и расстояния, соответствующие всем путям, ведущим к этому узлу.
Редюсер выберет самое короткое из этих расстояний, а затем обновит расстояние в структуре данных узла.

Очевидно, что параллельный поиск в ширину представляет собой итеративный алгоритм, где каждая итерация соответствует заданию MapReduce. При первом запуске алгоритма «обнаруживаются» все узлы, подключенные к источнику. На второй итерации обнаруживаются все узлы, связанные с ними, и так далее. Каждая итерация алгоритма расширяет «границу поиска» на один шаг, и в итоге все узлы будут обнаружены на кратчайших расстояниях (при условии, что граф полностью связный). Прежде чем рассмотреть завершение работы алгоритма, есть еще одна деталь, необходимая для того, чтобы алгоритм параллельного поиска в ширину работал. Необходимо как-то «передавать» структуру графа от одной итерации к другой. Это достигается путем создания самой структуры данных узла с идентификатором узла в качестве ключа (псевдокод, строка 4 в *Mapper*). В редукторе нужно отличать структуру данных узла от значений расстояния (псевдокод, строки 5–6 в *Reducer*) и обновлять минимальное расстояние в структуре данных узла перед тем, как выдать его в качестве конечного значения. Окончательный результат теперь готов служить входными данными для следующей итерации.

BFS: критерий завершения

- Как много итераций нужно для завершения параллельного BFS?
- Когда первый раз посетили искомую вершину, значит найден самый короткий путь
- Ответ на вопрос
  - Равно диаметру графа (наиболее удаленные друг от друга вершины)
- Правило шести рукопожатий?
- Практическая реализация
  - Внешняя программа-драйвер для проверки оставшихся вершин с дистанцией ∞
  - Можно использовать счетчики из Hadoop MapReduce

Алгоритм 2: Поиск кратчайшего пути на графе по алгоритму BFS

Реализация включает:

1. Формирование набора исходных данных: структура, содержащая списки смежности вершин исходного графа.
2. Указание начальной и искомой вершин пути на графе.
3. Код функции RECORDREADER, формирующей набор исходных данных для каждой итерации, в том числе и для первой итерации.
4. Код функций MAP и REDUCE для каждой итерации.
5. Код функции, формирующей признак завершения итераций.
6. Код цикла итераций алгоритма BFS, с вызовом модельной функции Mapreduce в теле цикла.
7. Вывод на печать кратчайшего пути на графе от исходной до искомой вершины.

Исходные данные в файле: **Lab1_данные.docx**

Результат выполнения лабораторной работы оформляется в виде файла блокнота Jupyter Notebook на основе шаблона **Lab1_MapReduce.ipynb**

Имя файла формируется по следующему шаблону:
613x_ФамилияИО_Lab1.ipynb

Файл результатов выполнения лабораторной работы направляется на электронную почту sepo@ssau.ru с произвольным сопровождающим текстом и темой письма:
613x ФамилияИО Lab1