Теория графов. Первая презентация

Зайцев Дмитрий

Введение

Остовное дерево — дерево, подграф исходного графа, имеющий такое же число вершин.

Минимальное остовное дерево — остовное дерево взвешенного графа, имеющее минимальный возможный вес.

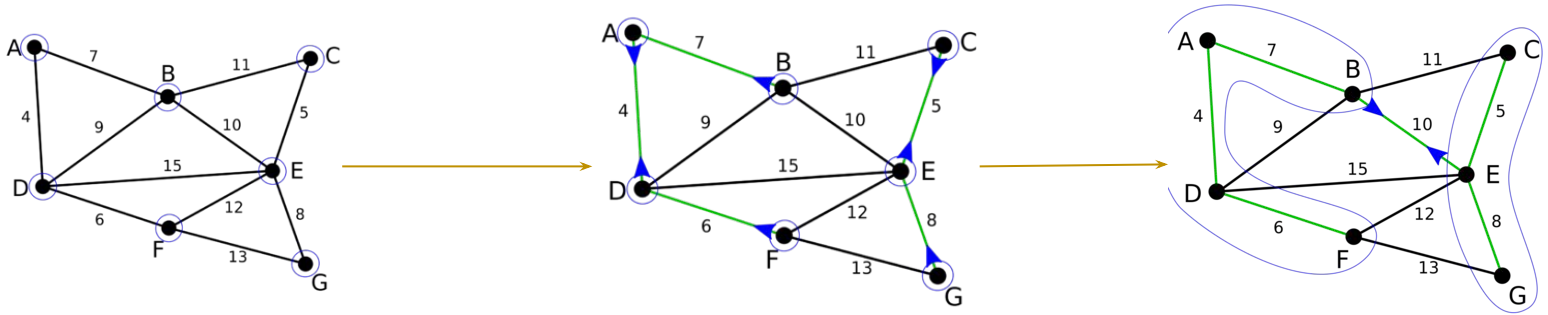
Алгоритм Борувки — алгоритм нахождения минимального остовного дерева в графе.

Алгоритм Борувки

ВХОД: Взвешенный граф G = (V, E).

ВЫХОД: Подграф F, минимальное остовное дерево.

1. Каждая вершина графа G — дерево, никакие ребра им не принадлежат.
2. Каждому дереву добавим минимальное инцидентное ему ребро.
3. Повторяем предыдущий шаг, пока не останется только одна компонента связности.



Характеристики вычислительной машины

- **Процессор**: AMD Ryzen 7 3800X (8 ядер, 16 потоков)
  - Базовая частота: 3900 MHz
  - L1-кэш: 512 KB
  - L2-кэш: 4 МБ
  - L3-кэш: 32 МБ
- **RAM**: 32 GB
- **GPU**: AMD Radeon RX 5700 XT
- **Операционная система**: Ubuntu 24.04.2

Датасет DIMACS 9th

Description	Nodes	Edges
Great Lakes	2,758,119	3,397,404
California and Nevada	1,890,815	2,315,222
Northeast USA	1,524,453	1,934,010
Northwest USA	1,207,945	1,410,387
Florida	1,070,376	1,343,951
Colorado	435,666	521,200
San Francisco Bay Area	321,270	397,415
New York City	264,346	365,050

Эксперименты

Эксперимент 1

Цель: Найти зависимость между плотностью графа и скоростью работы алгоритма на выбранной библиотеке.

Ход эксперимента:

1. Берём граф из датасета.
2. Начинаем *менять количество рёбер* в нём так, чтобы можно было построить график зависимости производительности библиотеки от плотности графа (1%, 3%, 5%, 7%, 10%, 13% и 15% от исходного числа рёбер).
3. Расширяем исходный датасет новыми графами.
4. Проверяем, будет ли наблюдаться зависимость для всех графов из датасета.

Эксперимент 1

Гипотеза: Нет однозначной связи между плотностью графов, полученных описанным выше методом и скоростью их обработки на выбранной библиотеке и для предложенного датасета.

Получение результатов:

- Выполняем 100 запусков алгоритма на каждом из графов в расширенном датасете.
- Находим среднее, стандартное отклонение и доверительные интервалы.
- Эти данные используем для проверки гипотезы.

Проверка гипотезы:

1. Для каждой группы графов ищем коэффициент корреляции (*коэффициент корреляции Пирсона*).
2. Для анализа используем α (уровень значимости) = 0.05.
3. Делаем поправку на множественную проверку гипотез.
4. Пытаемся опровергнуть гипотезу для каждого графа из исходного датасета.
5. Получаем табличку с результатами для каждого графа.

Эксперимент 1. Как будем менять графы

Какие рёбра лучше не трогать:

- Мосты, найдём их для каждого графа заранее.
- Рёбра с большими весами, скорее всего это важные дороги.
- С высокой мерой "центральности", они точно важны в графе.

Как это быстро это вычислить:

- Использовать python-igraph :)

Как контролировать, не испортили ли мы граф:

- Выполнять проверку на количество мостов, их не должно стать меньше
- Выполнять проверку на количество компонент связности, должна быть только одна
- Следить за количеством удалённых рёбер, у всех графов он должен быть одинаковый

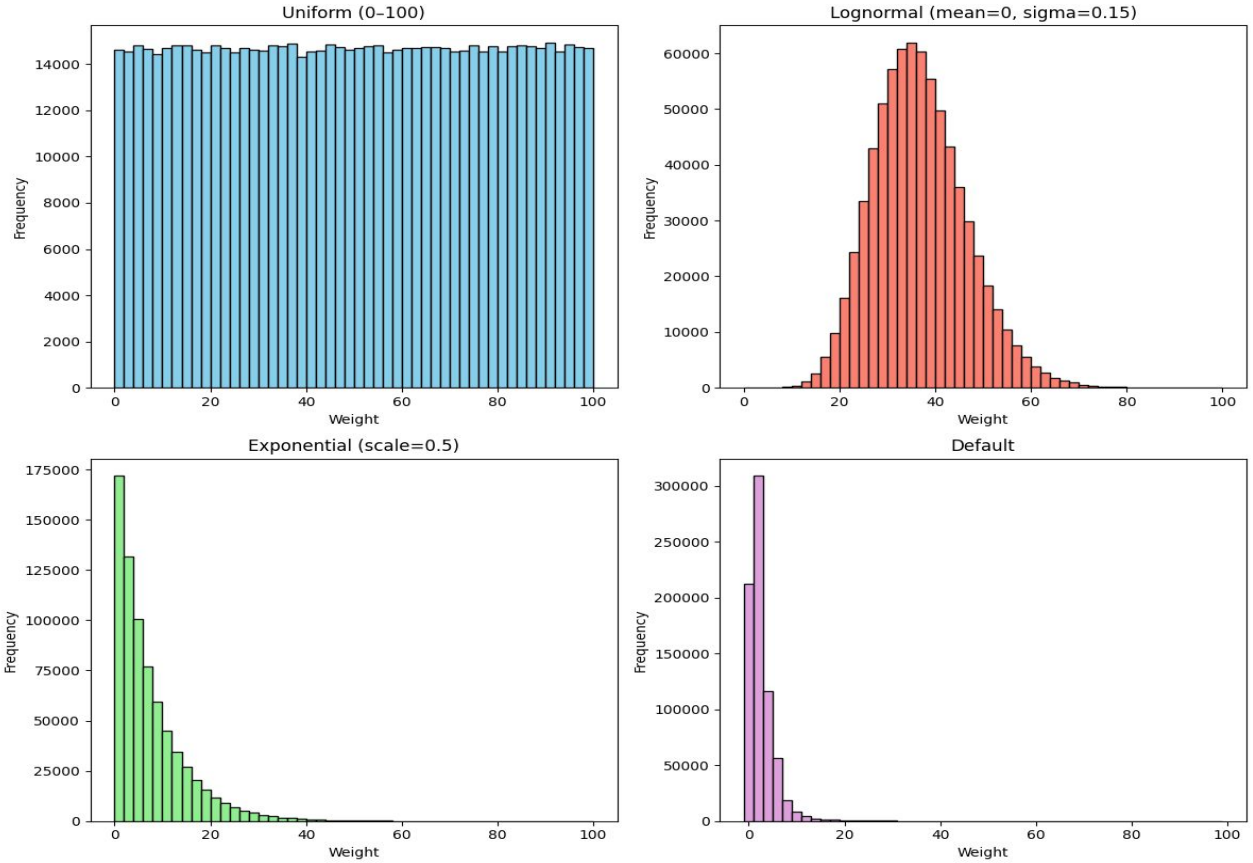
Эксперимент 2

Цель: Проанализировать производительность библиотеки на графах с одинаковой топологией, но разным распределением весов.

Ход эксперимента:

1. Берём граф из датасета.
2. Несколько раз меняем в нём распределение весов, но не трогаем топологию.
3. Расширяем исходный датасет новыми графами.
4. Проверяем, будет ли наблюдаться отличие в скорости обработки графов.

Эксперимент 2. Распределения



Эксперимент 2

Гипотеза: Нет никакой статистически значимой разницы в производительности выбранной библиотеки на графах с различным распределением весов, но полностью одинаковой топологией.

Получение результатов:

- Выполняем 100 запусков алгоритма на каждом из графов в обновлённом датасете.
- Находим среднее, стандартное отклонение и доверительные интервалы.
- Эти данные используем для проверки гипотезы.

Проверка гипотезы:

1. Для анализа используем α (уровень значимости) = 0.05.
2. Применяем однофакторный дисперсионный анализ для каждой группы графов.
3. Делаем поправку на множественную проверку гипотез.
4. Получаем табличку с результатами для каждого графа.

Реализация на GraphBlas

Борувка и линейная алгебра

Ребро — (вес, номер компоненты связности).
S — матрица смежности.
edge — вектор, содержащий минимальные рёбра инцидентные каждой вершине.
cedge — вектор, содержащий минимальные рёбра для потомков каждой вершины (только для корней деревьев).
parent — вектор, содержащий корень дерева для каждой вершины.

Combine — упаковывает вес и номер компоненты связности в единое значение (w, i).
Min — находит минимальное ребро.
CombMin — полукольцо из этих операций.

Борувка и линейная алгебра 2

Шаги:

1. Каждая вершина изначально является своим собственным родителем *parent[i] = i*.
2. Находим вектор *edge* как *S x parent*, используя операции полукольца.
3. Находим *cedge[parent[i]] = min(cedge[parent[i]], edge[i])*.
4. Обновляем *parent*, объединяя компоненты связности.
5. Удаляем из *S* рёбра, находящиеся в одной компоненте связности.
6. Повторяем пункты *2-5*, пока *S* не опустеет.

$$S = \begin{pmatrix} \infty & 5 & 3 & \infty & \infty \\ 5 & \infty & 2 & 6 & \infty \\ 3 & 2 & \infty & 4 & 8 \\ \infty & 6 & 4 & \infty & 1 \\ \infty & \infty & 8 & 1 & \infty \end{pmatrix}$$

$$\text{parent} = \begin{pmatrix} 0 & 1 & 2 & 3 & 4 \end{pmatrix}^T$$

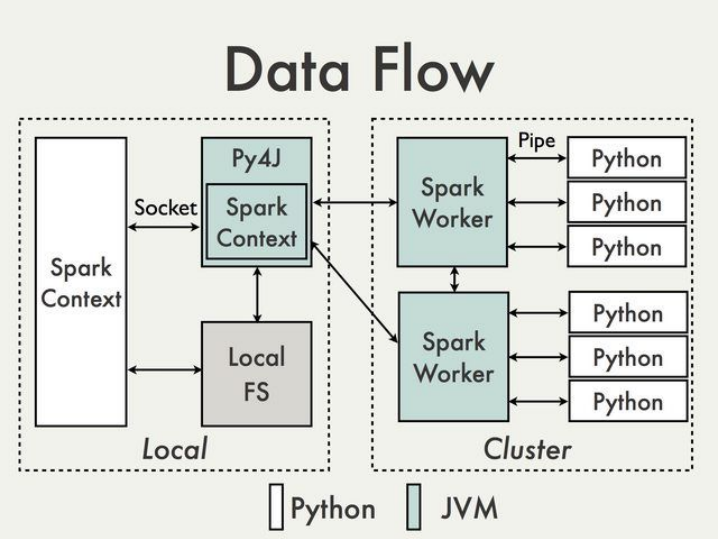
$$\begin{pmatrix} \infty \cdot 0 + 5 \cdot 1 + 3 \cdot 2 + \infty \cdot 3 + \infty \cdot 4 \\ 5 \cdot 0 + \infty \cdot 1 + 2 \cdot 2 + 6 \cdot 3 + \infty \cdot 4 \\ 3 \cdot 0 + 2 \cdot 1 + \infty \cdot 2 + 4 \cdot 3 + 8 \cdot 4 \\ \infty \cdot 0 + 6 \cdot 1 + 4 \cdot 2 + \infty \cdot 3 + 1 \cdot 4 \\ \infty \cdot 0 + \infty \cdot 1 + 8 \cdot 2 + 1 \cdot 3 + \infty \cdot 4 \end{pmatrix} = \begin{pmatrix} (3,2) \\ (2,2) \\ (2,1) \\ (1,4) \\ (1,3) \end{pmatrix}$$

Реализация на PySpark

PySpark

PySpark = Python API к Spark.

- При выполнении операций в PySpark, он создаёт объект, сериализует его и через *Py4J* передаёт в *JVM Spark*.
- Для работы с графами будем использовать *DataFrames, Transformations (select, join, filter, groupBy, union)* и *Actions (collect).* Только стандартные методы *PySpark*, без кастомных операций.
- Результаты операций будем хранить только в оперативной памяти и передавать их в Python-процесс только в последний момент.



Реализация на PySpark

Шаги:

1. Создание *SparkSession*. На вход получаем количество вершин и список всех рёбер с их весами.
2. Строим таблицу рёбер *(u, v, weight)*.
3. Строим таблицу соответствия *«вершина -> компонента связности»*, при инициализации каждая вершина это отдельная компонента.
4. Выполняем поиск рёбер, находящихся между разными компонентами связности, сохраняем их в виде таблицы (*u, v, weight, comp_u, comp_v*).
5. Для каждой компоненты находим минимальное выходящее из неё ребро и объединяем с другой компонентой, обновляя таблицу из пункта 3.
6. Повторяем шаги 4-5 пока у нас больше одной компоненты связности.