

# 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.

# Задачи и проблемы на графах

- Поиск кратчайшего пути
  - Роутинг траффика
  - Навигация маршрута
- Поиск Minimum Spanning Tree
  - Телекоммуникационные компании
- Поиск максимального потока (Max Flow)
  - Структура компьютеров и серверов Интернет
- Bipartite matching
  - Соискатели и работодатели
- Поиск “особых” вершин и/или групп вершин графа
  - Коммьюнити пользователей
- PageRank

# 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?

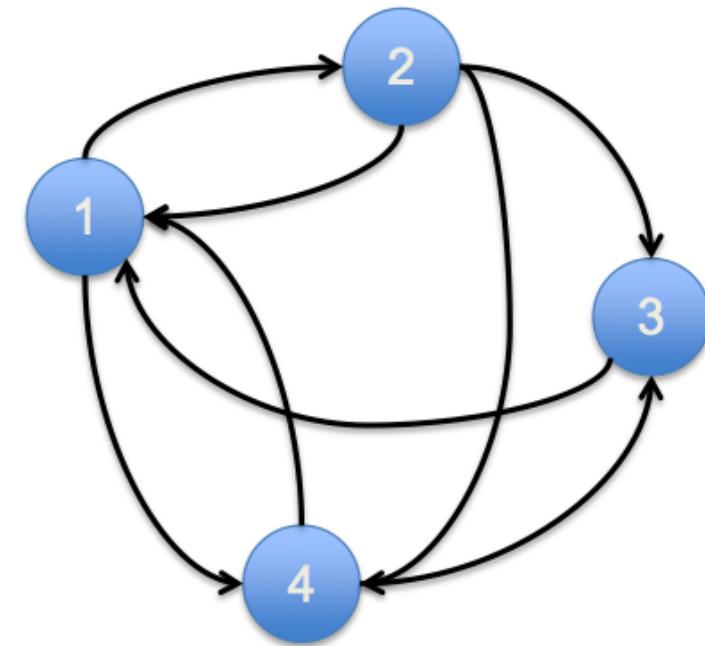
# Представление графов

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

# Матрица смежности

- Граф представляется как матрица  $M$  размером  $n \times n$ 
  - $n = |V|$
  - $M_{ij} = 1$  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)

# Алгоритм Дейкстры

- **Инициализация**

Метка искомой вершины полагается равной 0, метки остальных вершин — бесконечности.

Все вершины графа помечаются как непосещё

- **Шаг алгоритма**

Если все вершины посещены, алгоритм завершается.

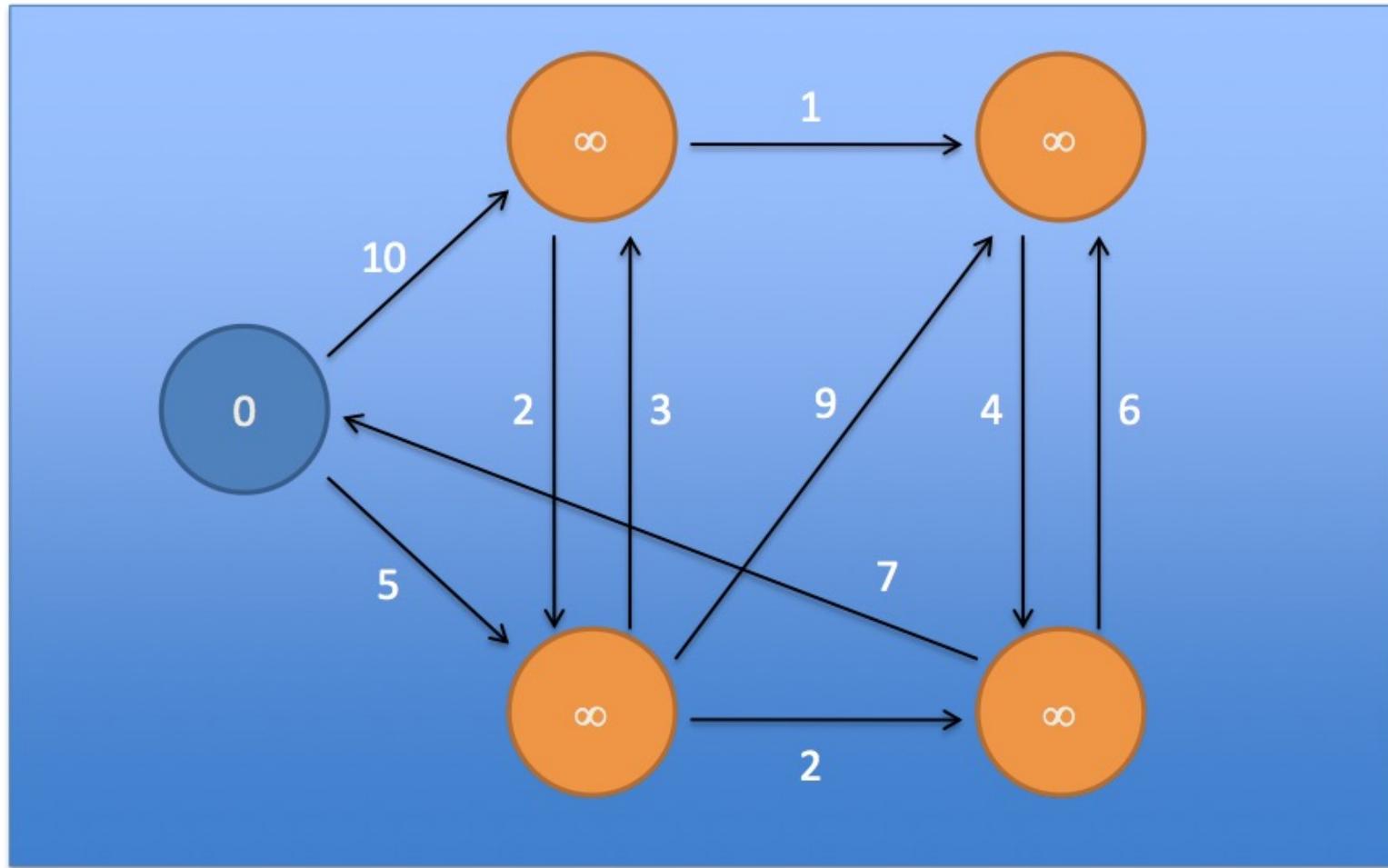
Из ещё не посещённых вершин выбирается вершина  $u$ , имеющая минимальную метку.

Для каждого соседа вершины  $u$ , кроме отмеченных как посещённые, рассмотрим новую длину пути, равную сумме значений текущей метки  $u$  и длины ребра, соединяющего  $u$  с этим соседом.

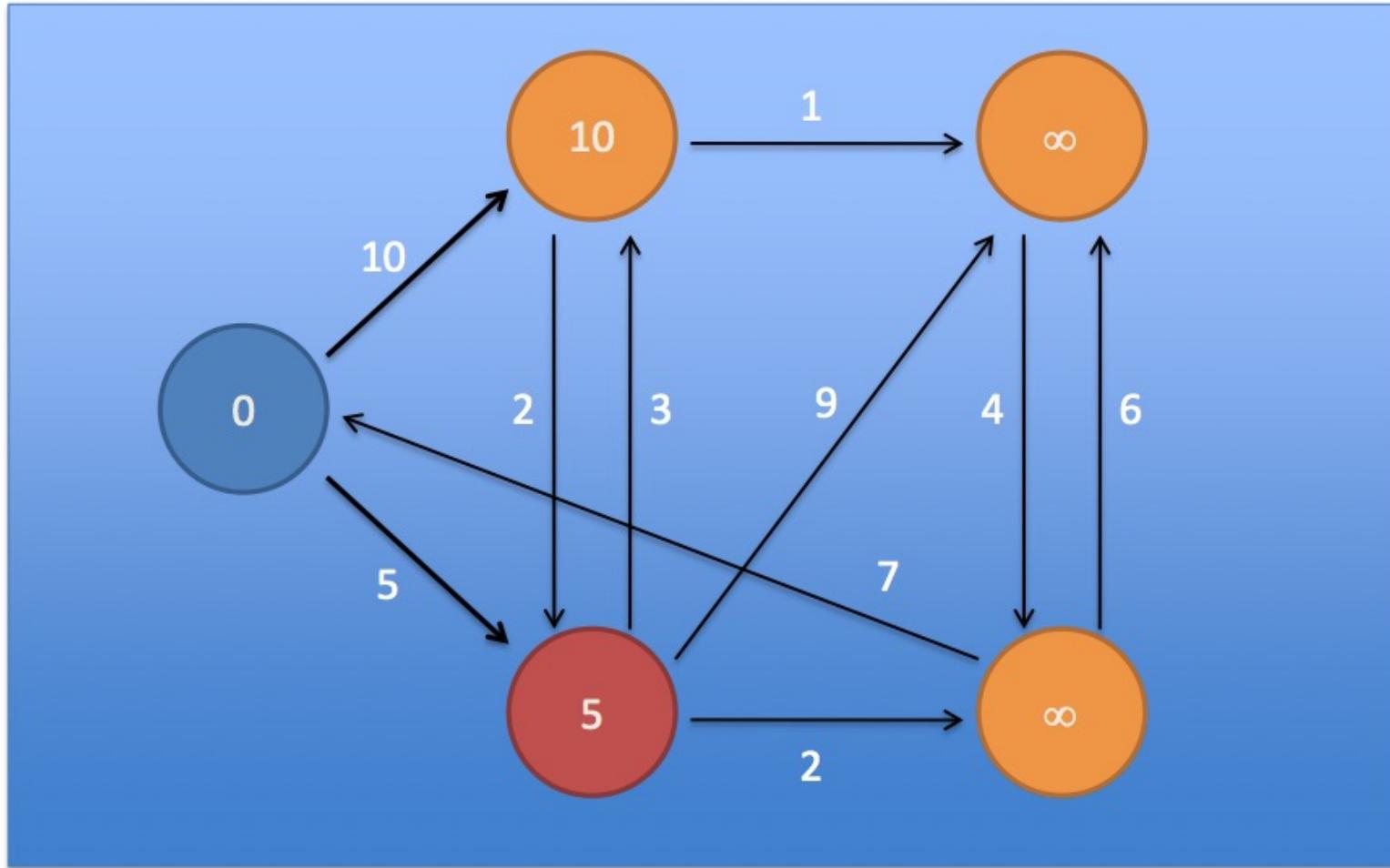
Если полученное значение длины меньше значения метки соседа, заменим значение метки полученным значением длины.

Рассмотрев всех соседей, пометим вершину  $u$  как посещённую и повторим шаг алгоритма.

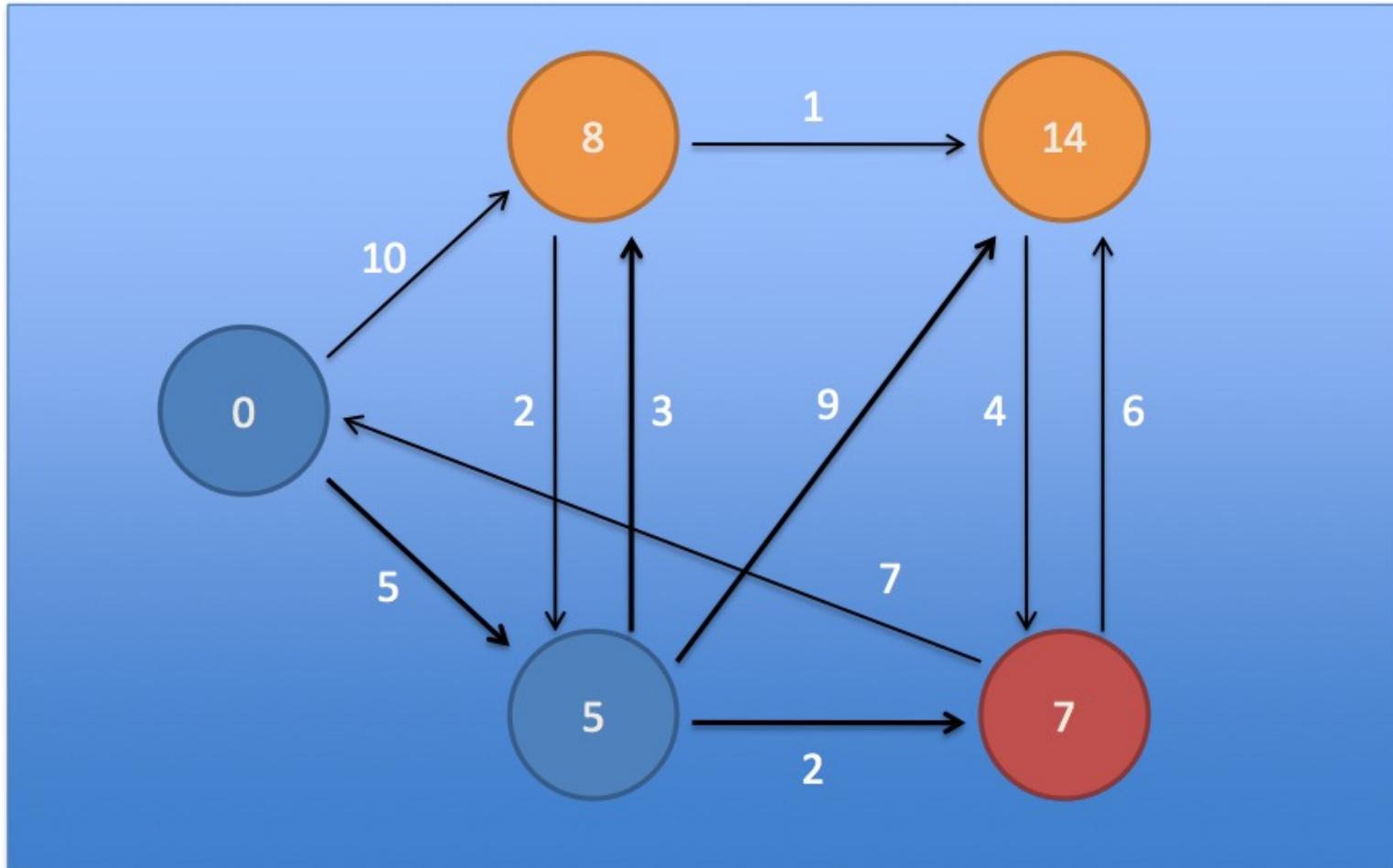
# Алгоритм Дейкстры (пример)



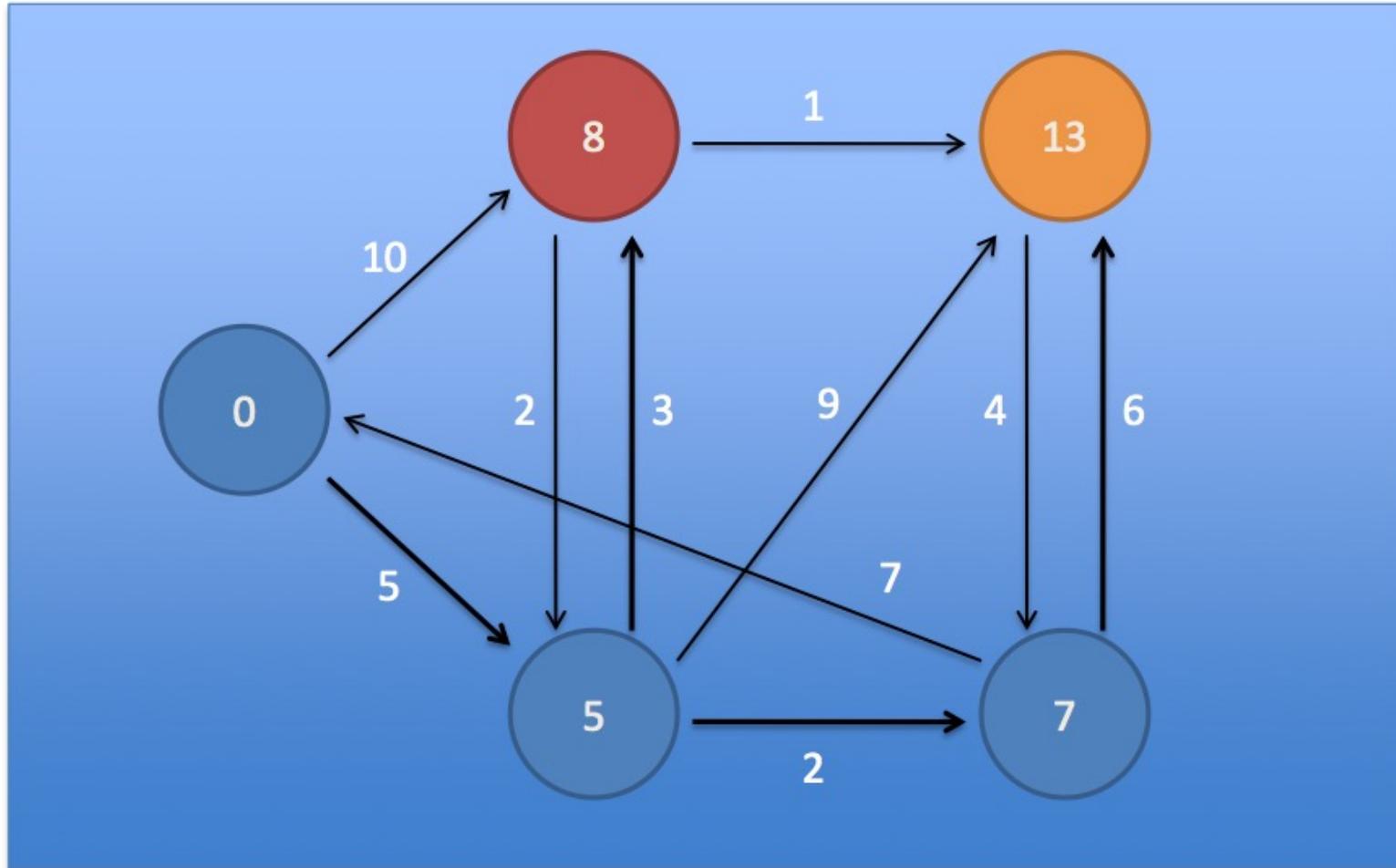
# Алгоритм Дейкстры (пример)



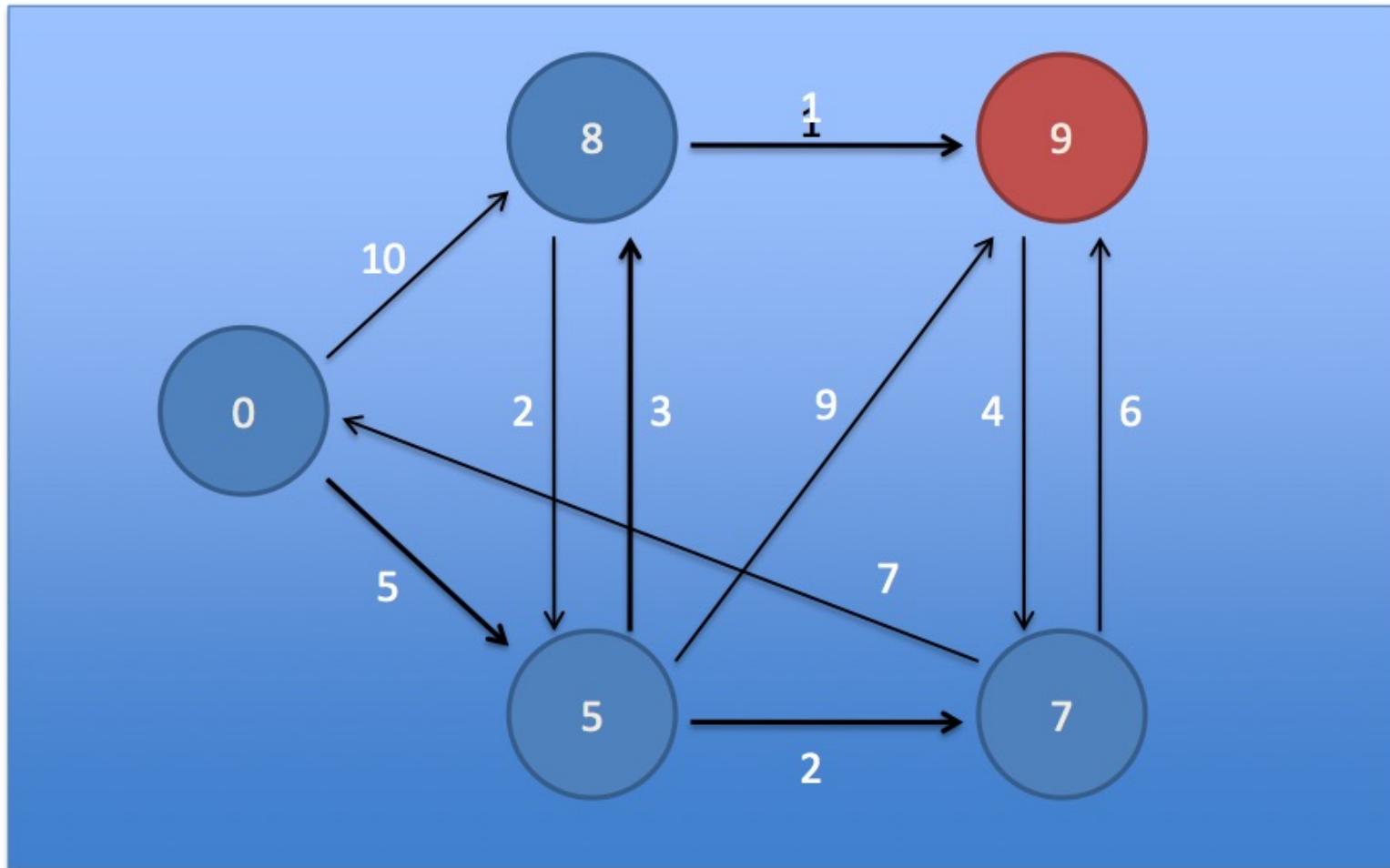
# Алгоритм Дейкстры (пример)



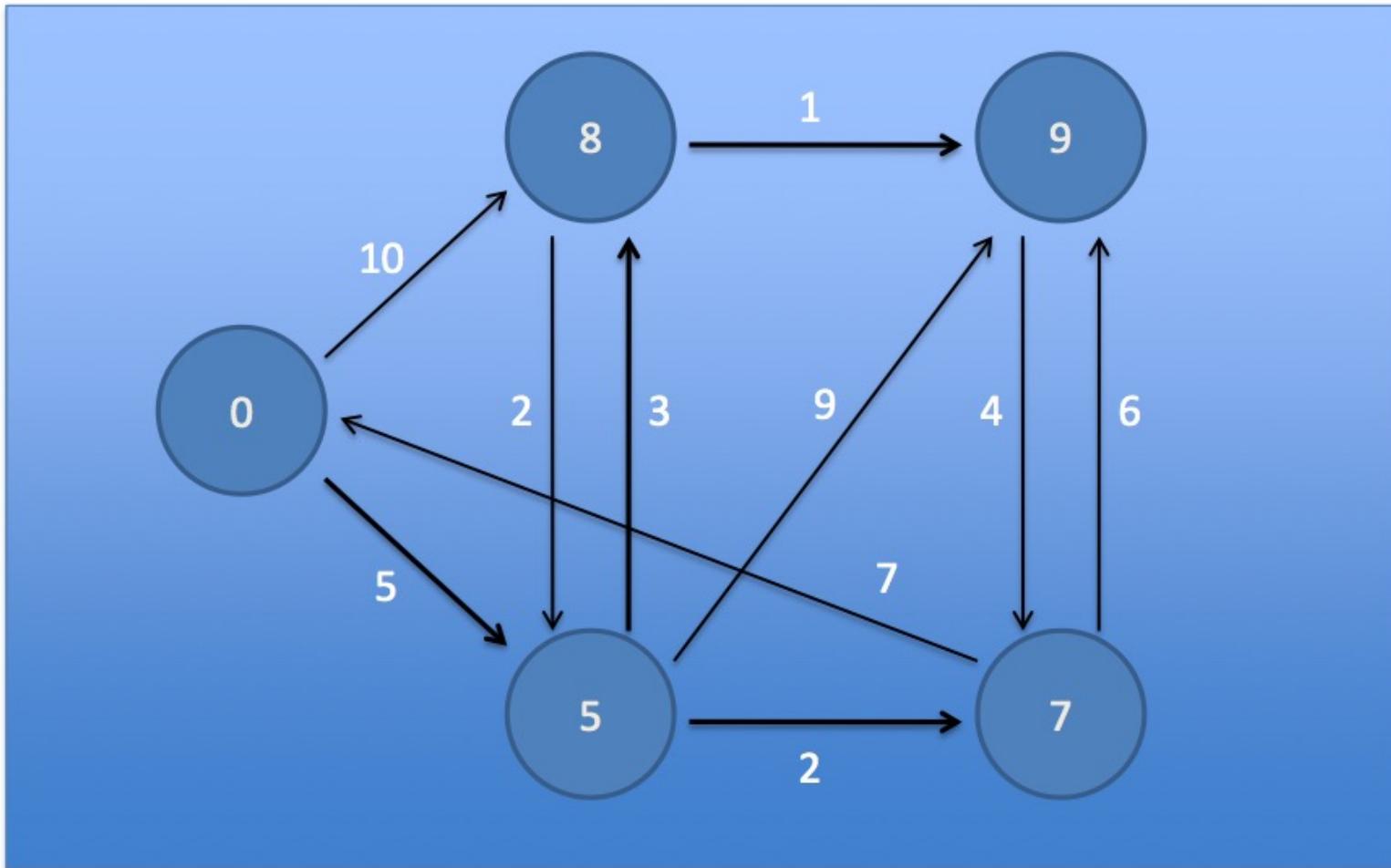
# Алгоритм Дейкстры (пример)



# Алгоритм Дейкстры (пример)



# Алгоритм Дейкстры (пример)

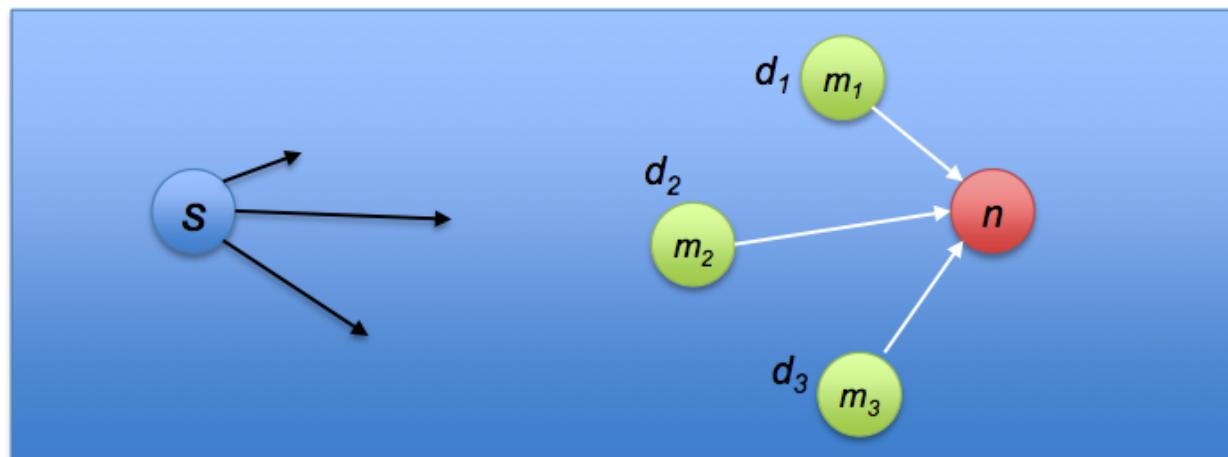


# Алгоритм Дейкстры

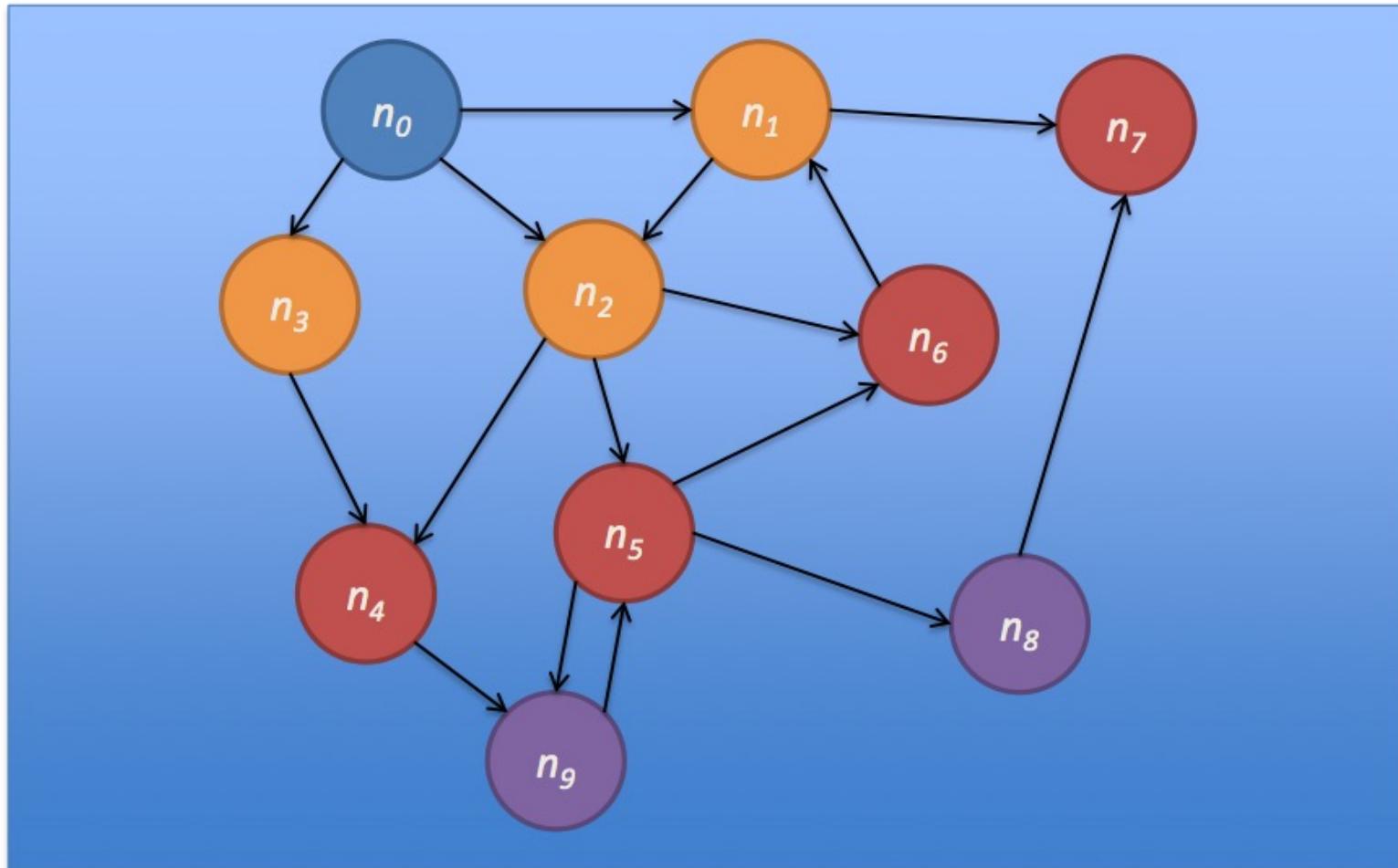
```
Dijkstra( $G, w, s$ )
     $d[s] \leftarrow 0$ 
    for all vertex  $v \in V$  do
         $d[v] \leftarrow \infty$ 
     $Q \leftarrow \{V\}$ 
    while  $Q = \emptyset$  do
         $u \leftarrow ExtractMin(Q)$ 
        for all vertex  $v \in u.AdjacencyList$  do
            if  $d[v] > d[u] + w(u, v)$  then
                 $d[v] \leftarrow d[u] + w(u, v)$ 
```

# 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  $M$   
 $DISTANCETo(n) = 1 + \min(DISTANCETo(m), m \in M)$



# 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:
  - $\forall m \in \text{adjacency list}: \text{emit } (m, d + 1)$
- 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)
```

# 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  $\infty$
  - Можно использовать счетчики из Hadoop MapReduce

# BFS vs Дейкстра

- Алгоритм Дейкстры более эффективен
  - На каждом шаге используются вершины только из пути с минимальным весом
  - Нужна дополнительная структура данных (*priority queue*)
- MapReduce обходит все пути графа параллельно
  - Много лишней работы (brute-force подход)
  - Полезная часть выполняется только на текущей границе обхода
- Можно ли использовать MapReduce более эффективно?

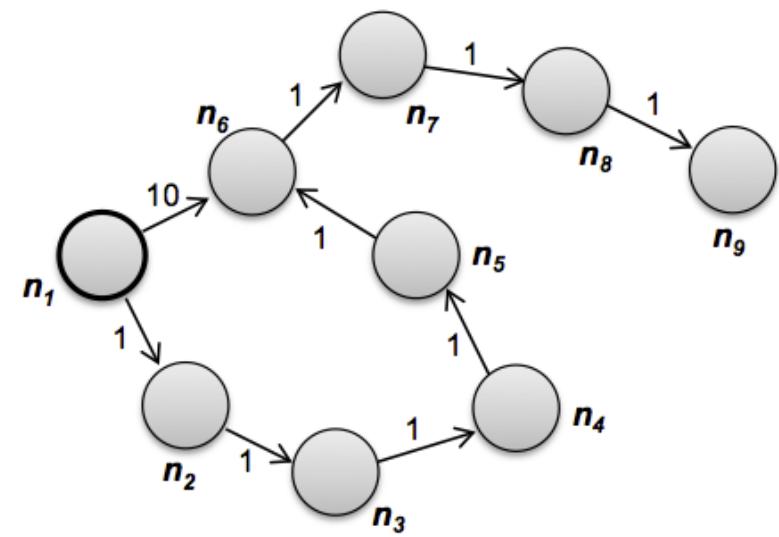
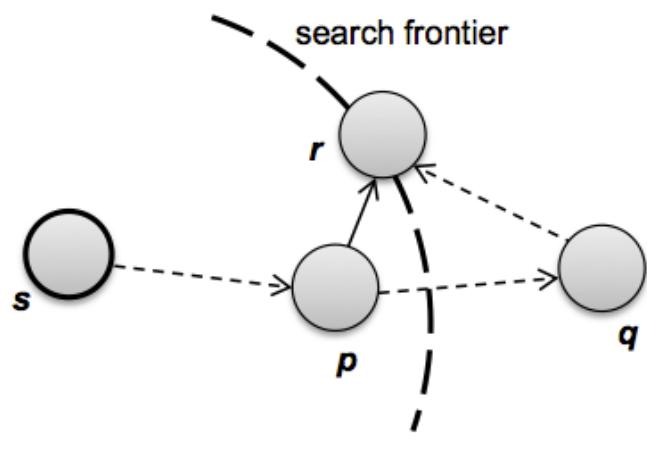
# BFS: Weighted Edges

- Добавим положительный вес каждому ребру
  - Почему вес ребра не может быть отрицательным?
- Простая доработка: добавим вес  $w$  для каждого ребра в список смежных вершин
  - В mapper, emit  $(m, d + w_p)$  вместо  $(m, d + 1)$  для каждой вершины  $m$
- И все?

# BFS Weighted: критерий завершения

- Как много итераций нужно для завершения параллельного BFS (взвешенный граф)?
- Когда первый раз посетили искомую вершину, значит найден самый короткий путь
- **И это неверно!**

# BFS Weighted: сложности



# BFS Weighted: критерий завершения

- Как много итераций нужно для завершения параллельного BFS (взвешенный граф)?
- В худшем случае:  $N - 1$
- В реальном мире  $\sim =$  диаметру графа
- Практическая реализация
  - Итерации завершаются, когда минимальный путь у каждой вершины больше не меняется
  - Для этого можно также использовать счетчики в MapReduce

# Графы и MapReduce

- Большое кол-во алгоритмов на графах включает в себя:
  - Выполнение вычислений, зависимых от фич ребер и вершины, на каждой вершине
  - Вычисления, основанные на обходе графа
- Основной рецепт:
  - Представлять графы в виде списка смежности
  - Производить локальные вычисления на маппере
  - Передавать промежуточные вычисления по исходящим ребрам, где ключом будет целевая вершина
  - Выполнять агрегацию на редьюсере по данных из входящих вершин
  - Повторять итерации до выполнения критерия сходимости, который контролируется внешним драйвером
  - Передавать структуру графа между итерациями

# Случайное блуждание по Web

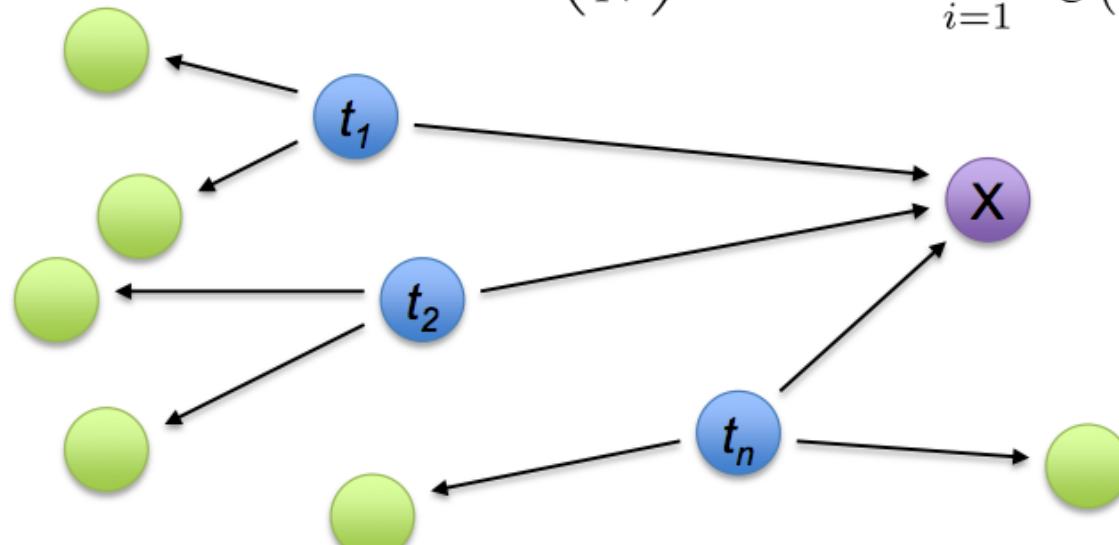
- Модель блуждающего веб-серфера
  - Пользователь начинает серфинг на случайной веб-странице
  - Пользователь произвольно кликает по ссылкам, тем самым перемещаясь от страницы к странице
- PageRank
  - Характеризует кол-во времени, которое пользователь провел на данной странице
  - Математически – это распределение вероятностей посещения страниц
- PageRank определяет понятие важности страницы
  - Соответствует человеческой интуиции?
  - Одна из тысячи фич, которая используется в веб-поиске

# PageRank, определение

Дана страница  $x$ , на которую указывают ссылки  $t_1 \dots t_n$ , где

- $C(t)$  степень out-degree для  $t$
- $\alpha$  вероятность случайного перемещения (*random jump*)
- $N$  общее число вершин в графе

$$PR(x) = \alpha \left( \frac{1}{N} \right) + (1 - \alpha) \sum_{i=1}^n \frac{PR(t_i)}{C(t_i)}$$



# Вычисление PageRank

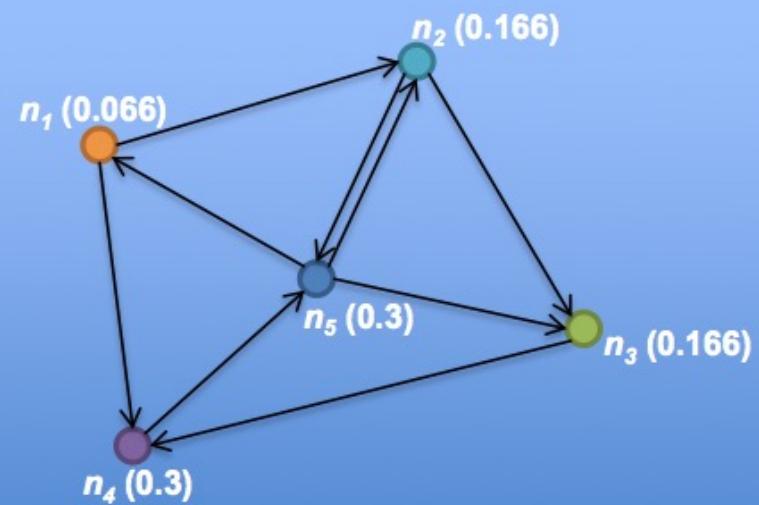
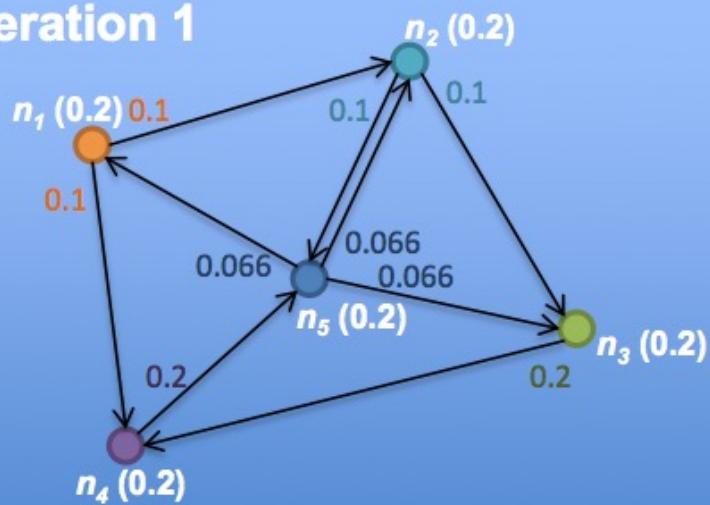
- Свойства PageRank'a
  - Может быть рассчитан итеративно
  - Локальный эффект на каждой итерации
- Набросок алгоритма
  - Начать с некоторыми заданными значениями  $PR_i$ ,
  - Каждая страница распределяет  $PR_i$ , “кредит” всем страниц, на которые с нее есть ссылки
  - Каждая страница добавляет весь полученный “кредит” от страниц, которые на нее ссылаются, для подсчета  $PR_{i+1}$
  - Продолжить итерации пока значения не сойдутся

# Упрощения для PageRank

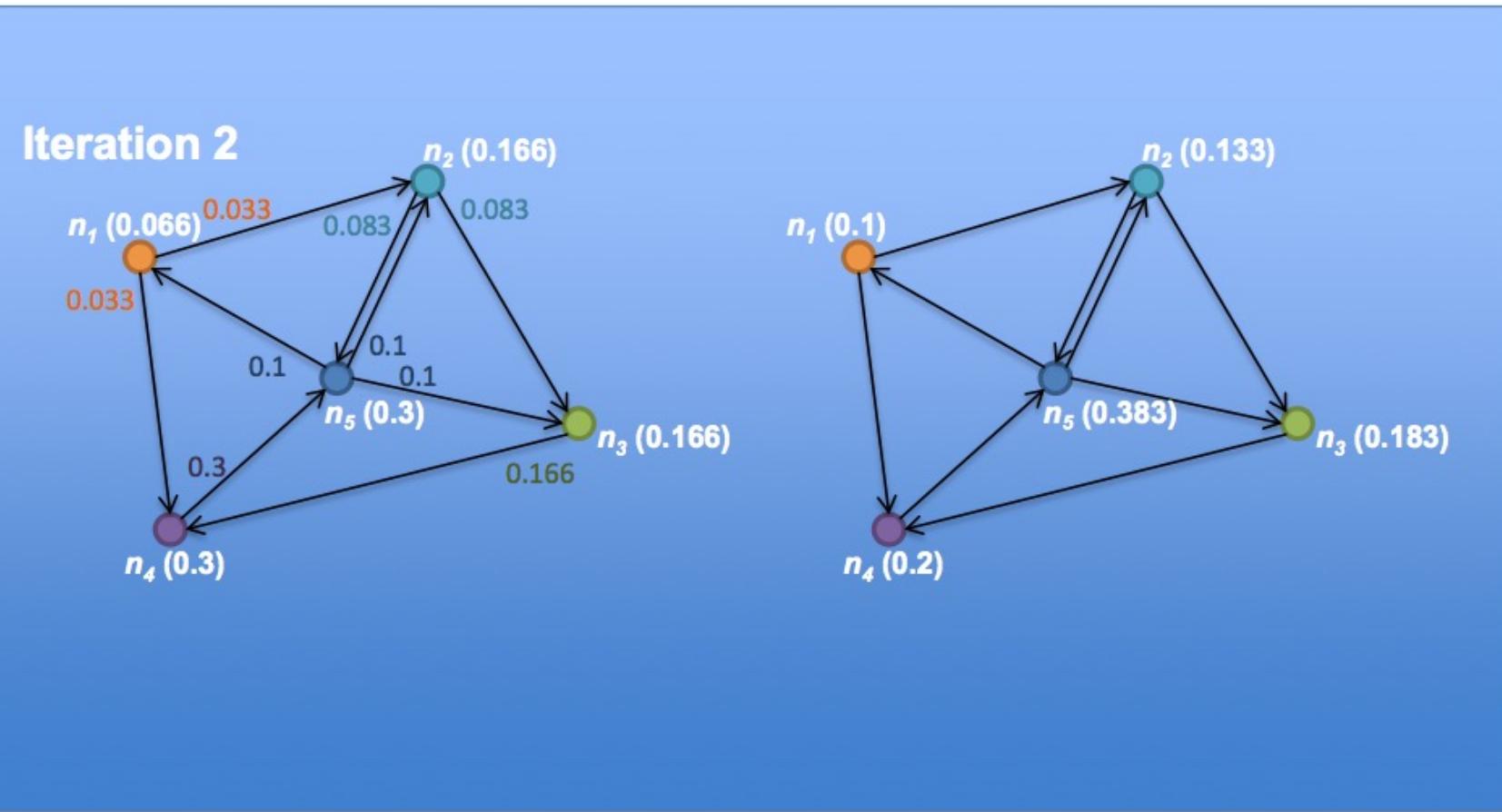
- Для начала рассмотрим простой случай
  - Нет фактора случайного перехода (*random jump*)
  - Нет “подвисших” вершин
- Затем, добавим сложностей
  - Зачем нужен случайный переход?
  - Откуда появляются “подвисшие” вершины?

# Пример расчета PageRank (1)

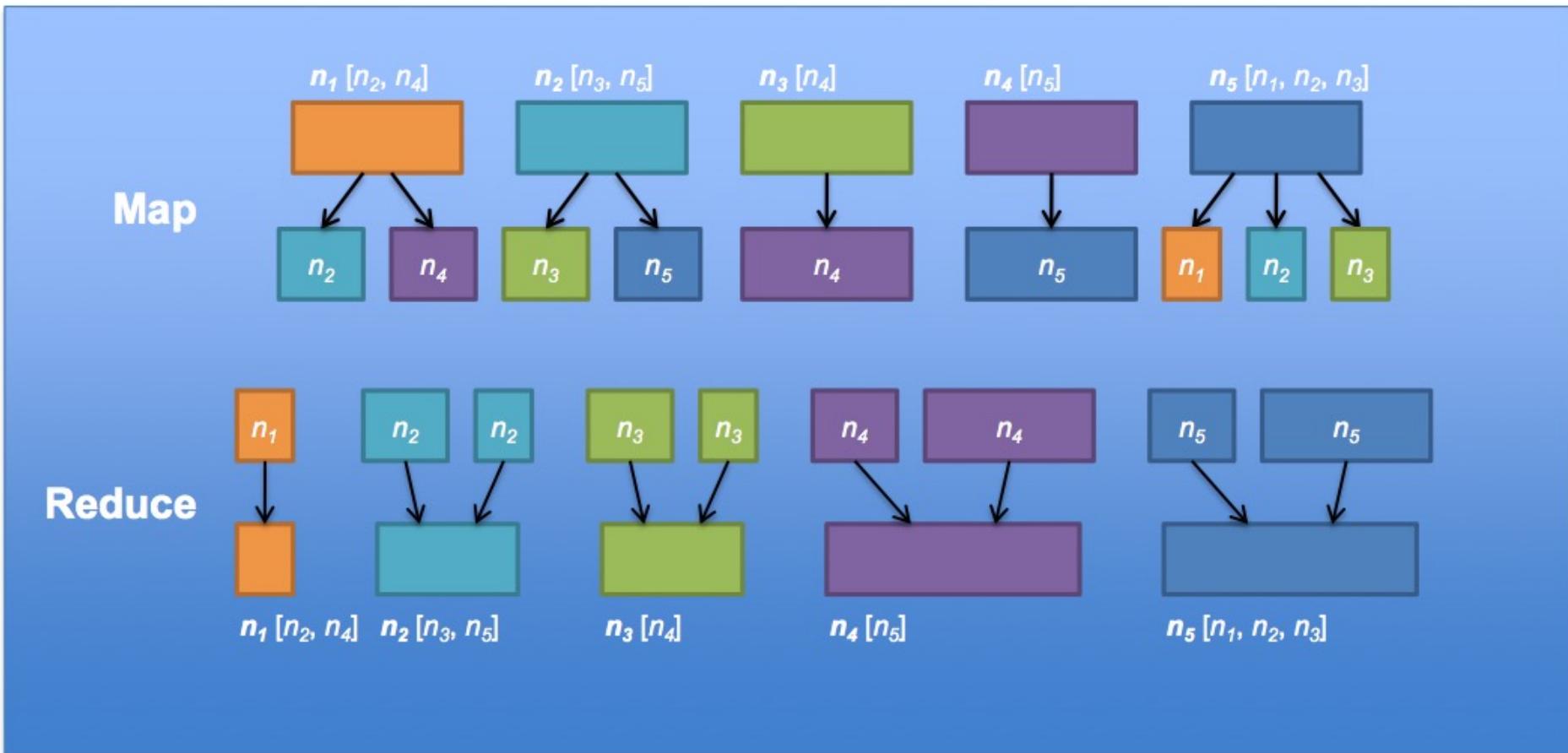
Iteration 1



# Пример расчета PageRank (2)



# PageRank на MapReduce



# PageRank: псевдокод

```
class Mapper
    method Map(nid n, node N)
         $p \leftarrow N.\text{PageRank} / |N.\text{AdjacencyList}|$ 
        Emit(nid n, N) // Pass along graph structure
        for all nodeid m  $\in N.\text{AdjacencyList}$  do
            Emit(nid m, p) // Pass PageRank mass to neighbors

class Reducer
    method Reduce(nid m, [p1, p2, ...])
        M  $\leftarrow \emptyset$ 
        for all p  $\in$  counts [p1, p2, ...] do
            if IsNode(p) then
                M  $\leftarrow p$  // Recover graph structure
            else
                s  $\leftarrow s + p$  // Sum incoming PageRank contributions
        M.PageRank  $\leftarrow s$ 
        Emit(nid m, node M)
```

# Полный PageRank

- Две дополнительные сложности
  - Как правильно обрабатывать “подвешенные” вершины?
  - Как правильно определить фактор случайного перехода (*random jump*)?
- Решение :
  - Второй проход для перераспределения “оставшегося” PageRank и учитывания фактор случайного перехода
$$p' = \alpha \left( \frac{1}{N} \right) + (1 - \alpha) \left( \frac{m}{N} + p \right)$$
    - $p$  – значение PageRank полученное “до”,  $p'$  – обновленное значение PageRank
    - $N$  - число вершин графа
    - $m$  – “оставшийся” PageRank
- Дополнительная оптимизация: сделать за один проход!

# Сходимость PageRank

- Альтернативные критерии сходимости
  - Продолжать итерации пока значения PageRank не перестанут изменяться
  - Продолжать итерации пока отношение PageRank не перестанут изменяться
  - Фиксированное число итераций

# Приложение PageRank

- Статическая приоритизация для веб-ранжирования
- Определение “особых вершин” в сети
- Рекомендация ссылок
- Дополнительная фича в решении какой-либо задачи machine learning

# Другие классы проблем на графах

- Поиск паттерна для подграфа
- Расчет простой статистки для графа
  - Распределение степени ребер по вершинам
- Расчет более сложной статистики
  - Коэффициенты кластеризации
  - Подсчет треугольников

## Основные проблемы для алгоритмов на графах

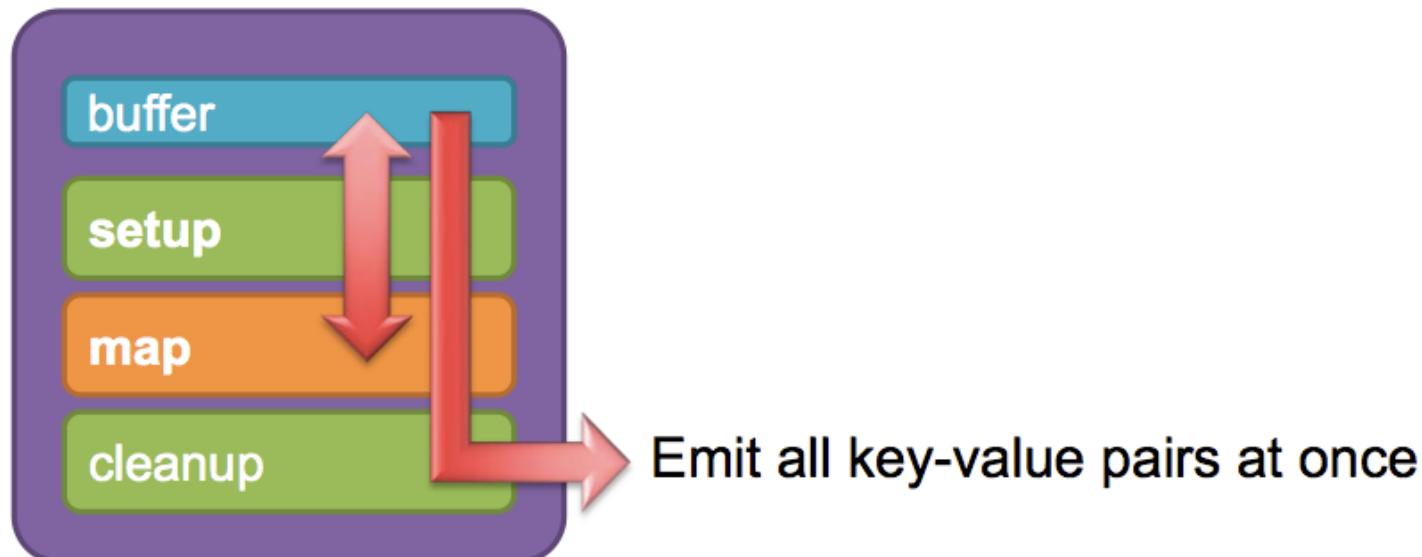
- Большие разреженные графы
- Топология графов

# MapReduce для графов – ложка дегтя

- Многословность Java
- Время запуска таска в Hadoop
- Медленные или зависшие таски
- Бесполезность фазы shuffle для графов
- Проверки на каждой итерации
- Итеративные алгоритмы на MapReduce неэффективны!

# In-Mapper Combining

- Использование комбайнеров
  - Выполнять локальную агрегацию на стороне map output
  - Минус: промежуточные данные все равно обрабатываются
- Лучше: *in-mapper combining*
  - Сохранять состояние между множеством вызовов map, агрегировать сообщения в буфер, писать содержимое буфера в конце
  - Минус: требуется управление памятью



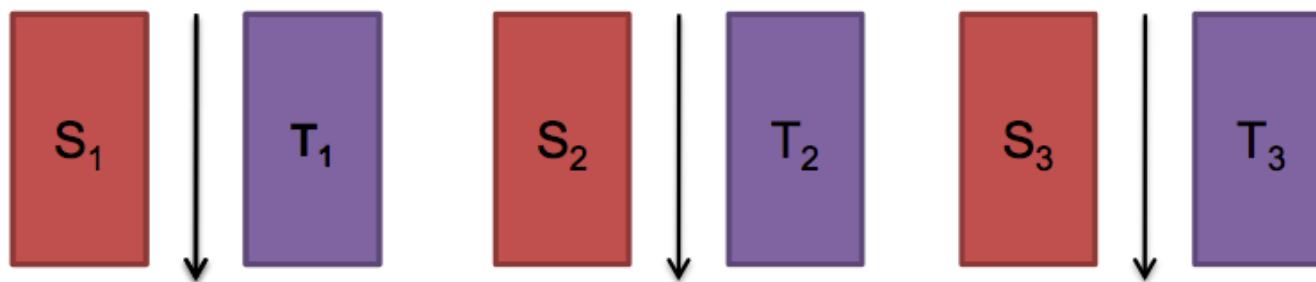
# Улучшение партиционирования

- По-умолчанию: hash partitioning
  - Произвольно присвоить вершину к партиции
- Наблюдение: много графов имеют локальную структуру
  - Напр., коммьюнити в соц.сетях
  - Лучшее партиционирование дает больше возможностей для локальной агрегации
- К сожалению, партиционирование довольно **сложно!**
  - Порой, это проблема курицы и яйца
  - Но иногда простые эвристики помогают
  - Для веб-графа: использовать партиционирование на основе домена от URL

# Schimmy Design Pattern

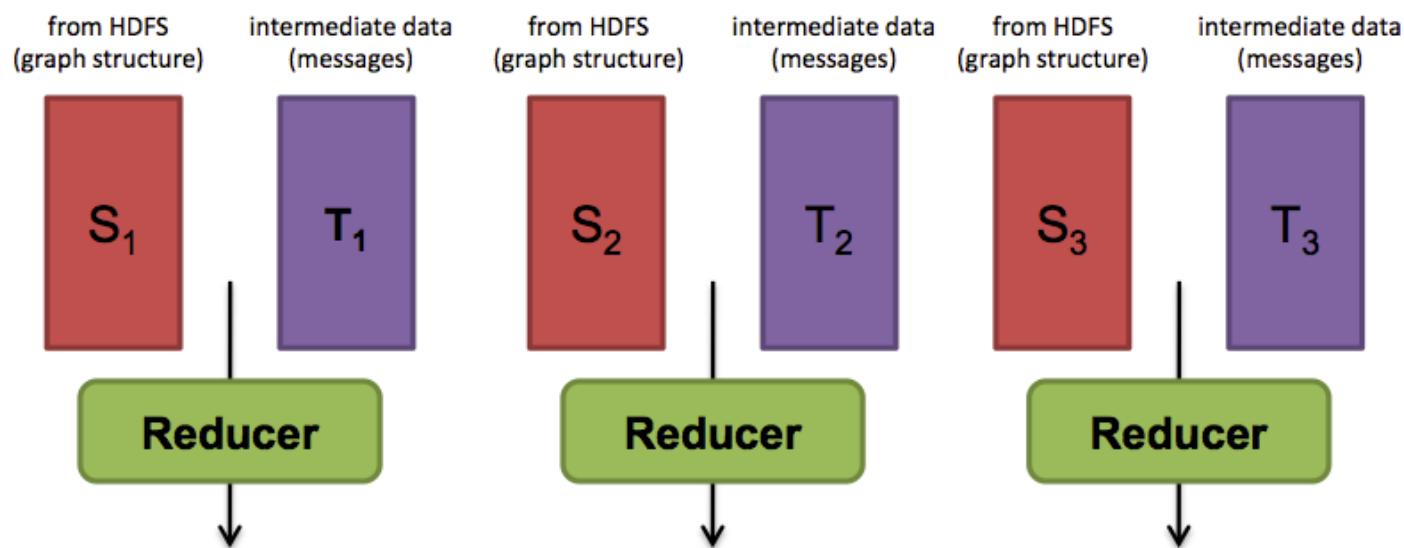
- Основная реализация содержит два набора данных:
  - *Messages* (актуальные вычисления)
  - *Graph structure* (структура обрабатываемого графа)
- Schimmy: разделить два набора данных, выполнять *shuffle* только для *messages*
  - Основная идея: выполнять *merge join* для *graph structure* и *messages*

обе части сортированы по *join key*, части консистентно партиционированы и сортированы по *join key*



# Используем Schimmy

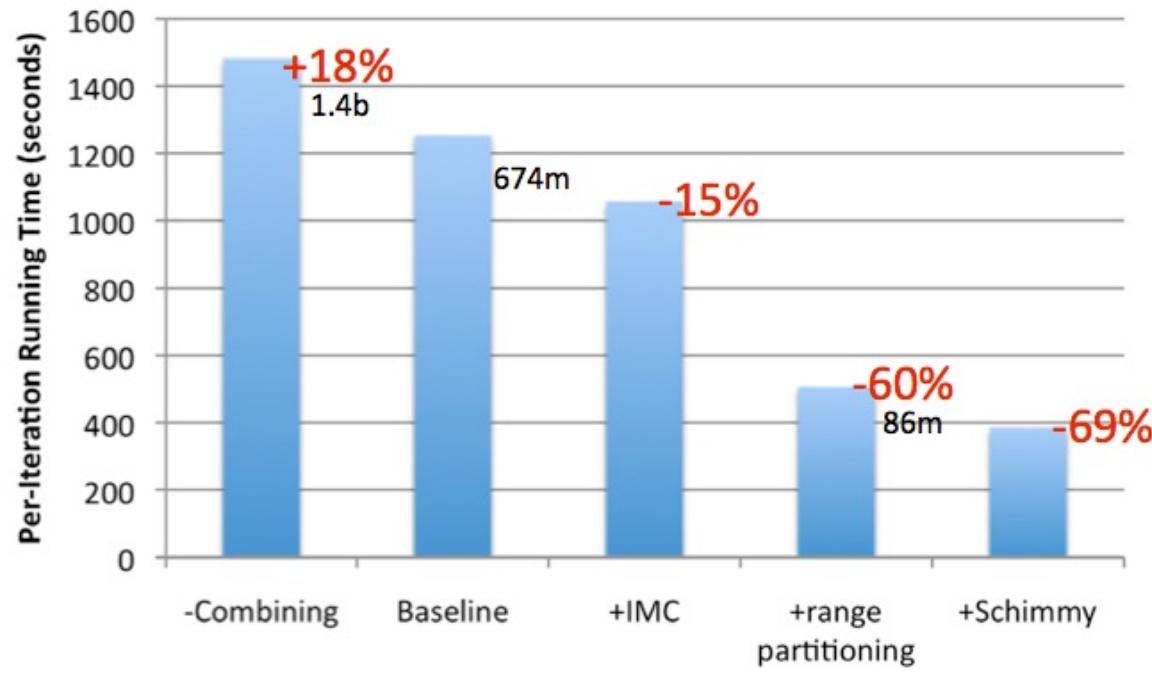
- Schimmy = на редьюсерах выполняется параллельный *merge join* между *graph structure* и *messages*
  - Консистентное партиционирование между входным и промежуточными данными (*intermediate data*)
  - *Mappers* пишут только *messages* (актуальные вычисления)
  - *Reducers* читают *graph structure* напрямую из HDFS



# Эксперимент

- Cluster setup:
  - 10 workers, each 2 cores (3.2 GHz Xeon), 4GB RAM, 367 GB disk
  - Hadoop 0.20.0 on RHELS 5.3
- Dataset:
  - Первый сегмент английского текста их коллекции ClueWeb09
  - 50.2m web pages (1.53 TB uncompressed, 247 GB compressed)
  - Extracted webgraph: 1.4 Млрд ссылок, 7.0 GB
  - Dataset сортирован в порядке краулинга
- Setup:
  - Измерялось время выполнения по каждой итерации (5 итераций)
  - 100 партиций

# Результаты





## Data-Intensive Text Processing with MapReduce

Jimmy Lin  
Chris Dyer

*SYNTHESIS LECTURES ON  
HUMAN LANGUAGE TECHNOLOGIES*

Graeme Hirst, Series Editor

### **Data-Intensive Text Processing with MapReduce**

Jimmy Lin and Chris Dyer (Authors) (April, 2010)

### **Data-Intensive Text Processing with MapReduce**

Jimmy Lin and Chris Dyer (Authors) (January 27, 2013)

<https://lintool.github.io/MapReduceAlgorithms/index.html>