ТЕХНОСФЕРА

Методы распределенной обработки больших объемов данных в Hadoop

Лекция 6: MapReduce в Hadoop, графы

Москва 2014

План

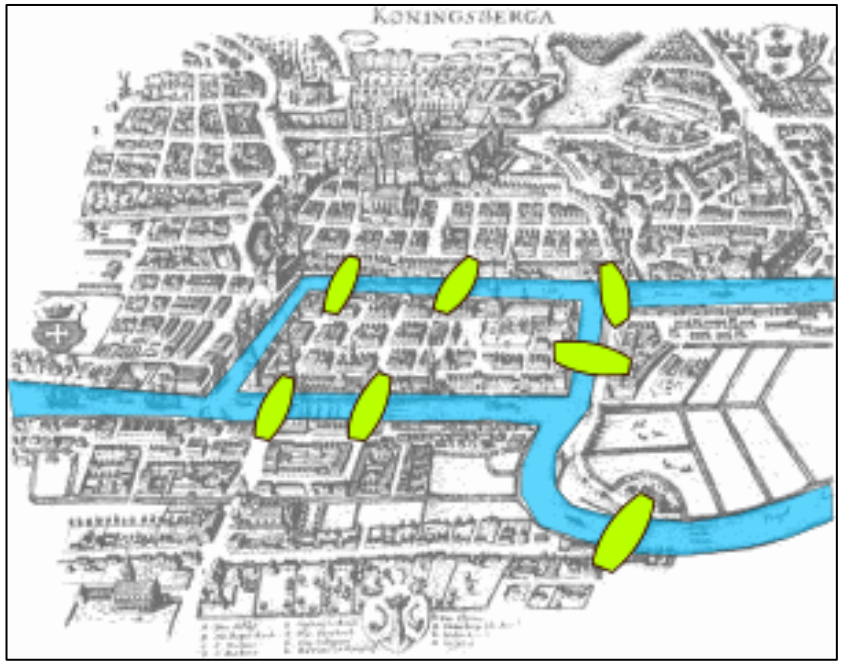
- Представление графов и связанные задачи
- Параллельный breadth-first search
- PageRank
- Не только PageRank: другие алгоритмы
- Оптимизация алгоритмов на графах

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 “особенных” вершин и/или групп вершин графа
  - Коммьюнити пользователей
- PageRank


KONINGSBERGA

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?

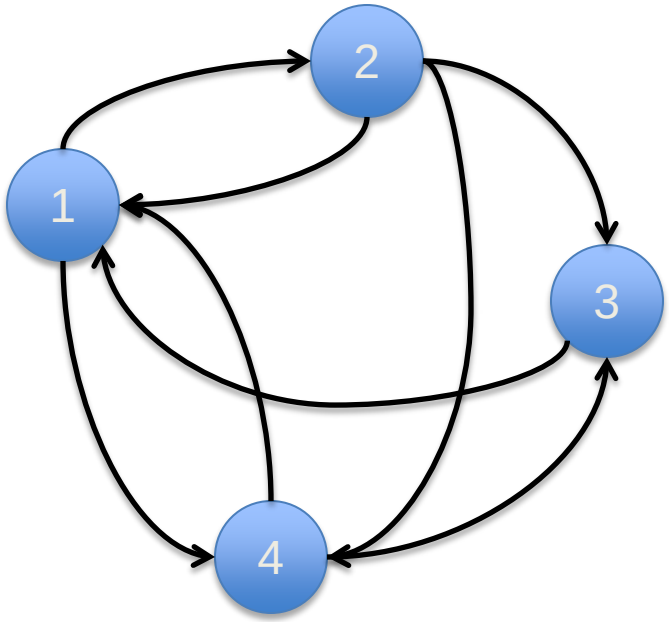
Представление графов

- 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

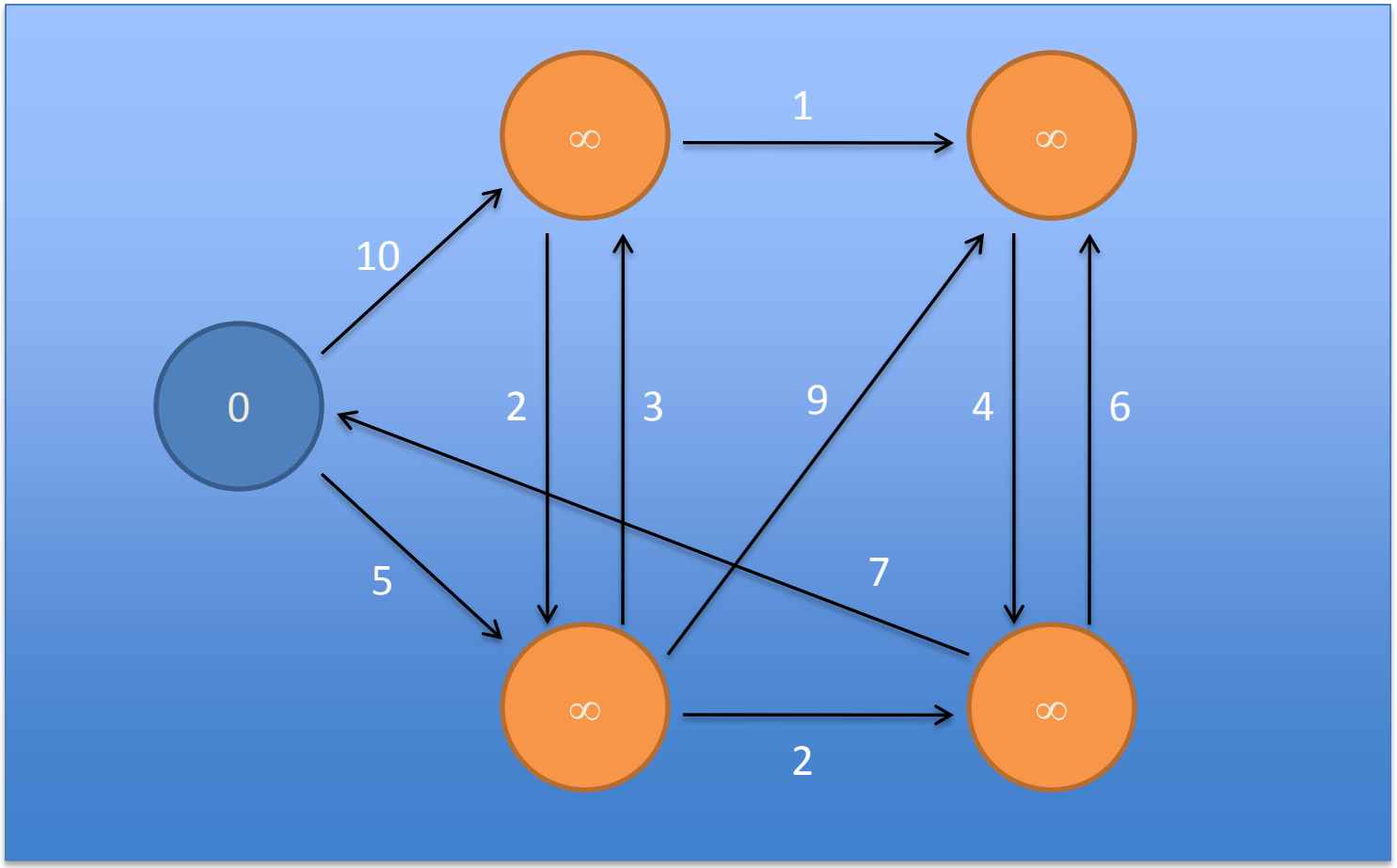
Списки смежности

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

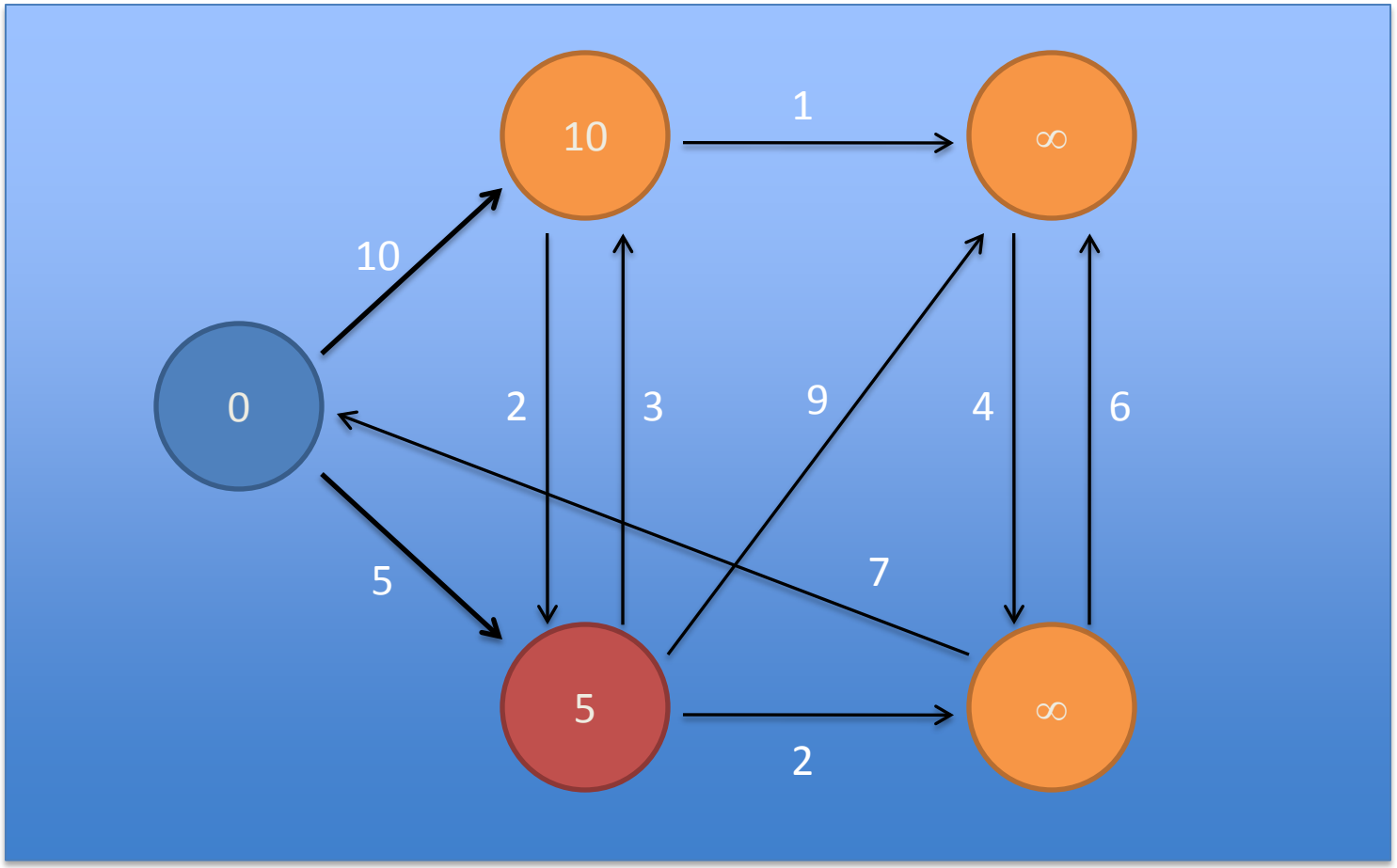
Поиск кратчайшего пути

- 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)

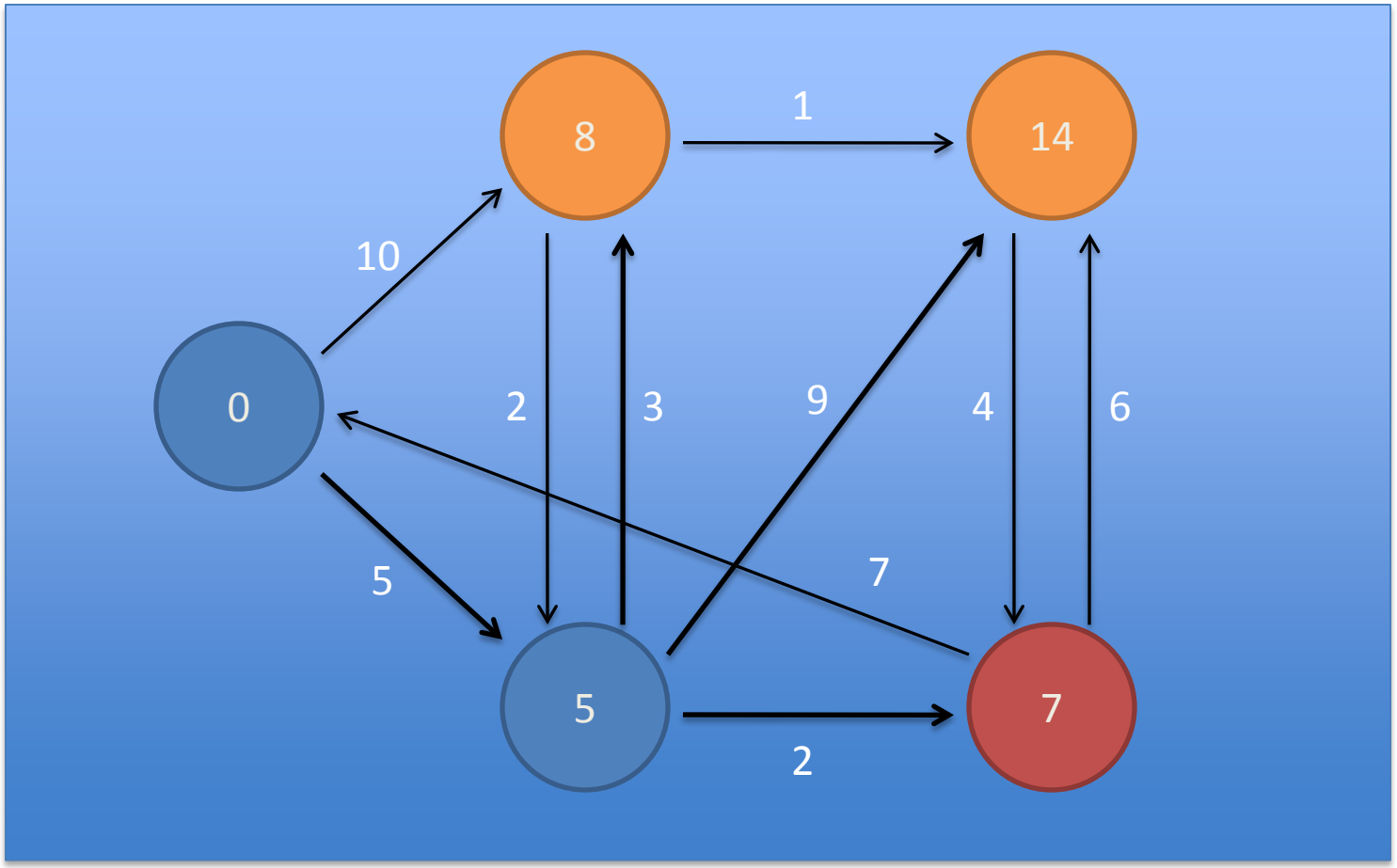
Алгоритм Дейкстры (пример)



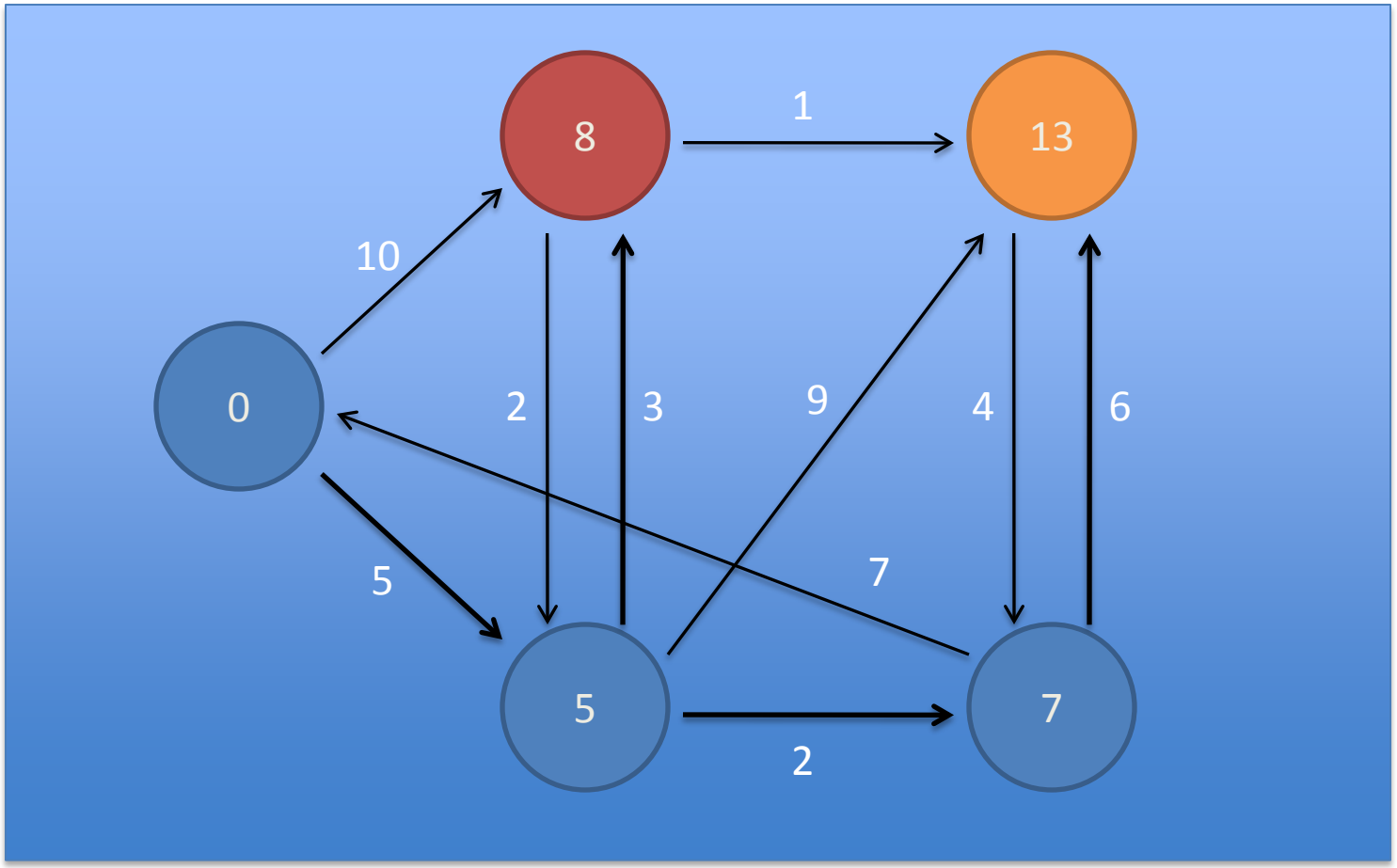
Алгоритм Дейкстры (пример)



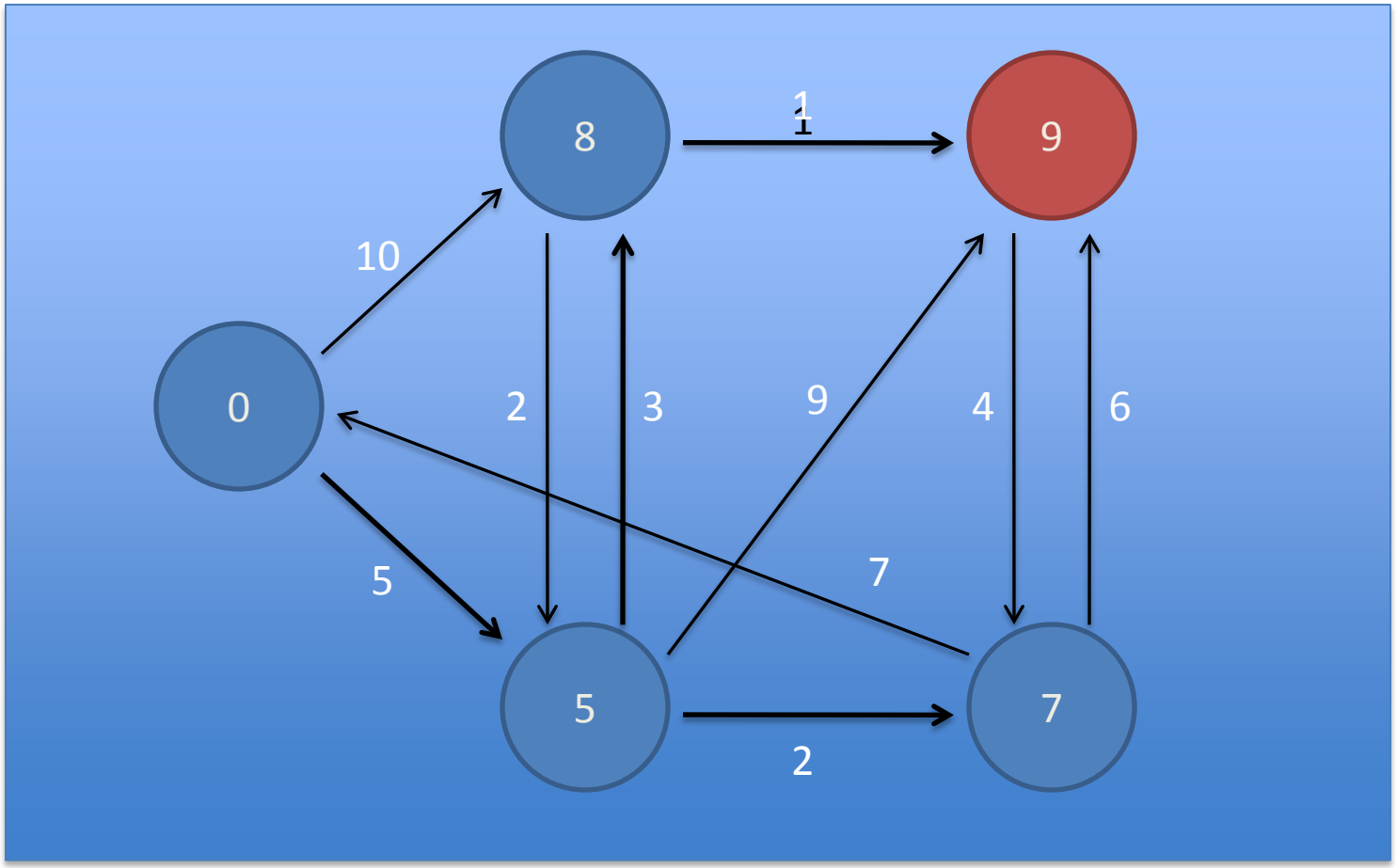
Алгоритм Дейкстры (пример)



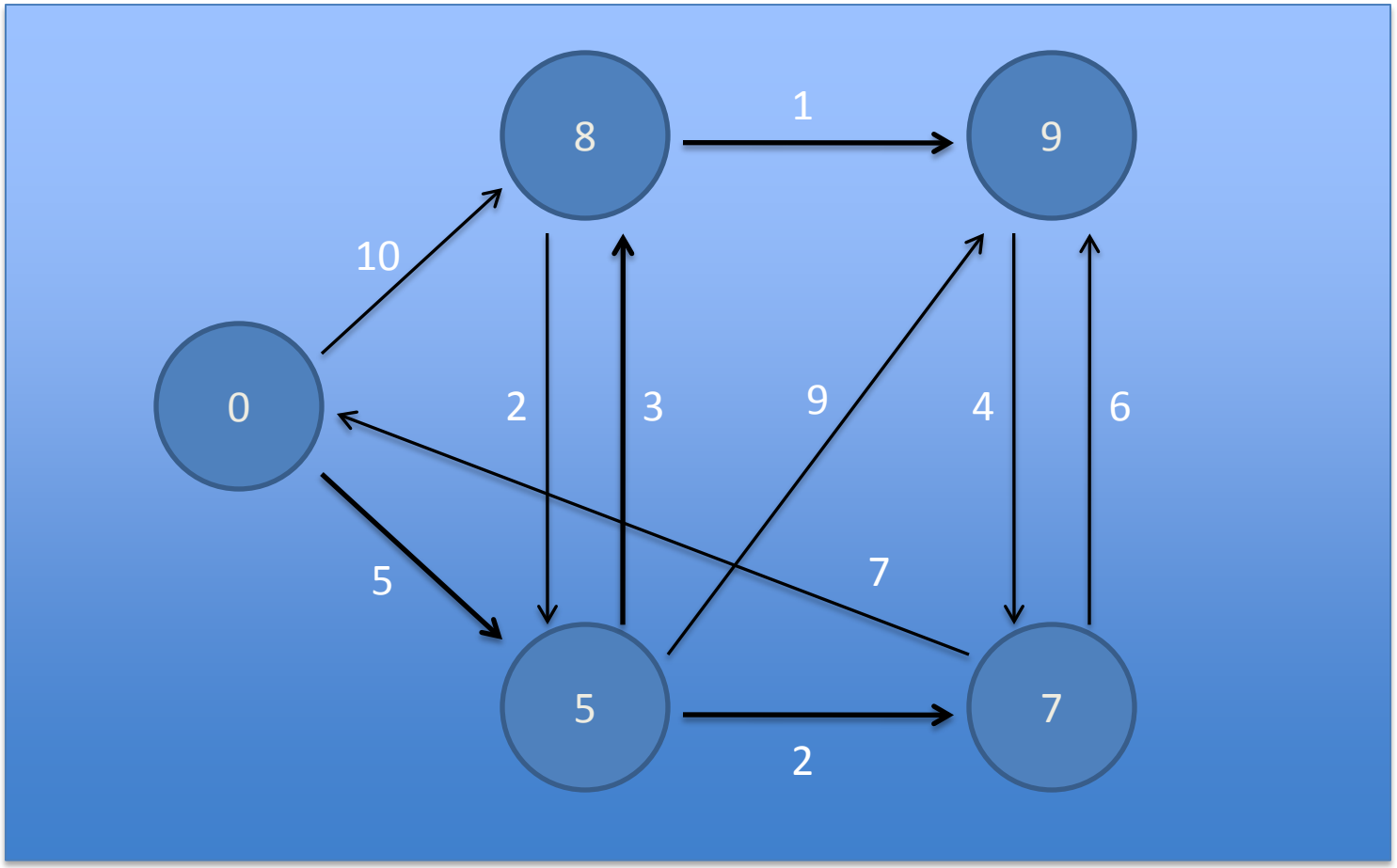
Алгоритм Дейкстры (пример)



Алгоритм Дейкстры (пример)



Алгоритм Дейкстры (пример)

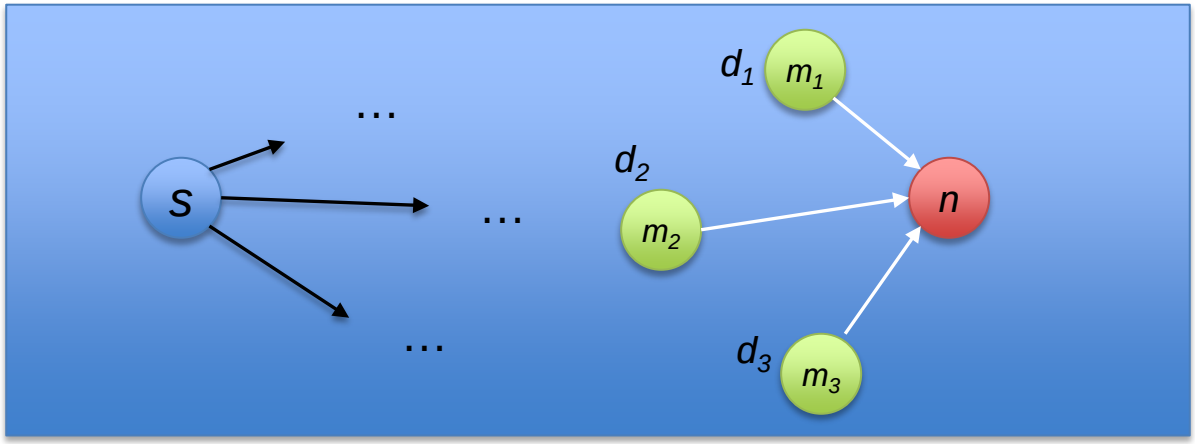


Алгоритм Дейкстры

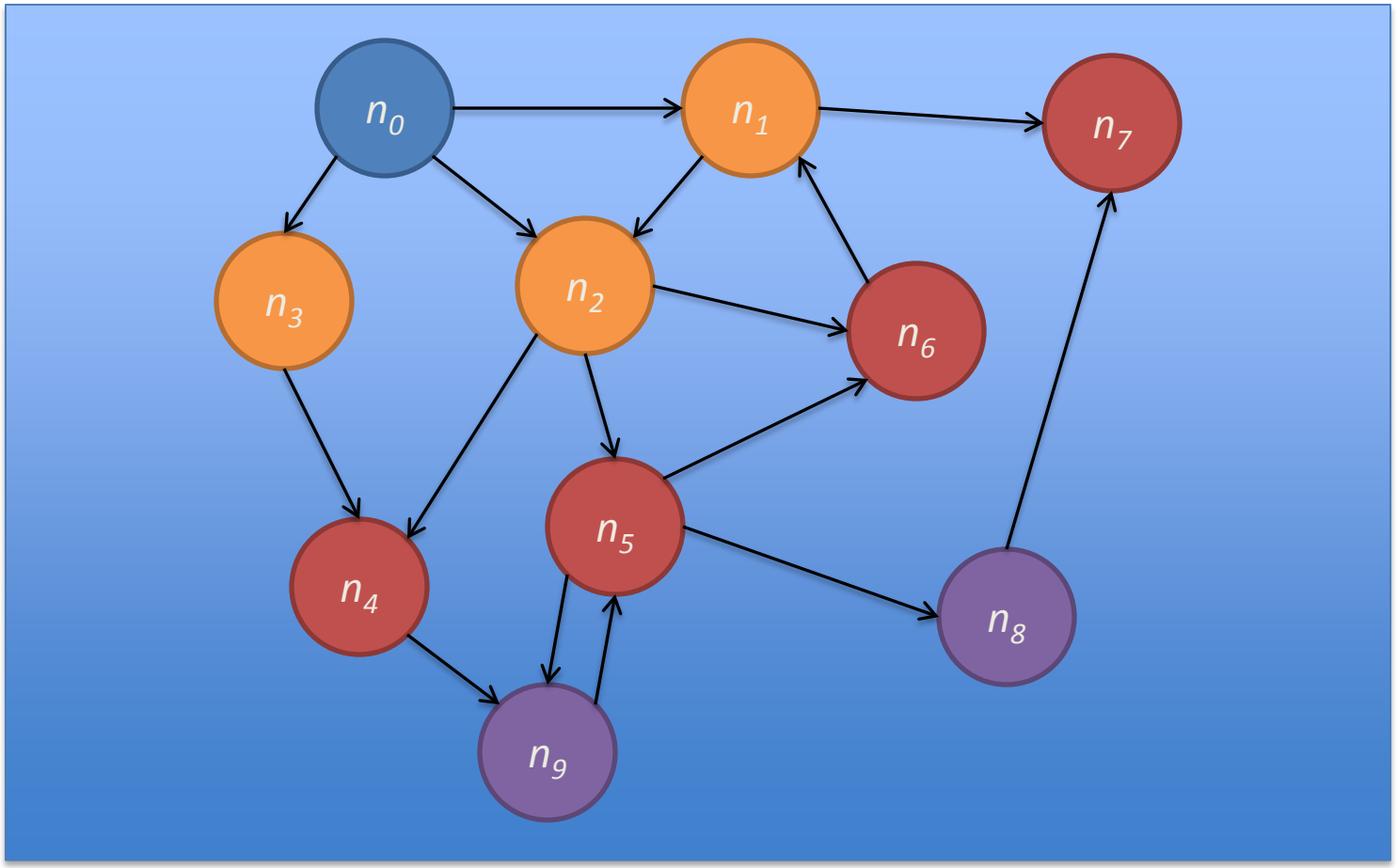
```
Dijkstra(G,w, s)
    d[s] ← 0
    for all vertex v ∈ V do
        d[v] ← ∞
    Q ← {V }
    while Q = ∅ do
        u ←ExtractMin(Q)
        for all vertex v ∈ u.AdjacencyList do
            if d[v] > d[u] + w(u, v) then
                d[v] ← d[u] + w(u, v)
```

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 ∈ 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)
```

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

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?

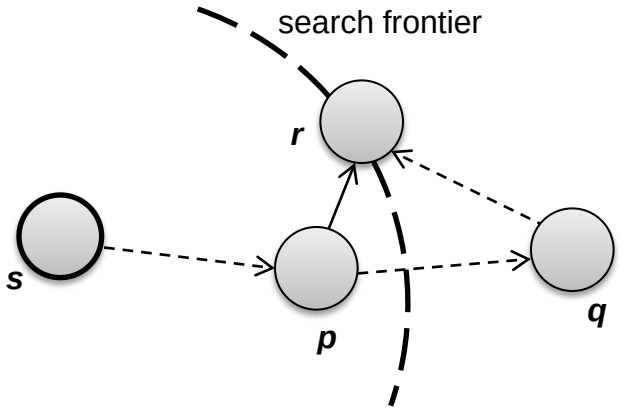
BFS: Weighted Edges

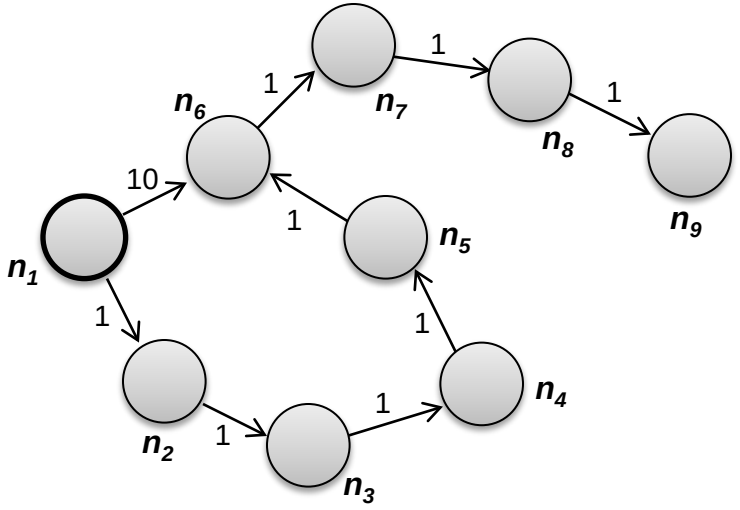
- Добавим положительный вес каждому ребру
  - Почему вес ребра не может быть отрицательным?
- Простая доработка: добавим вес *w* для каждого ребра в список смежных вершин
  - В mapper, emit (m, $d + w_p$) вместо (m, $d + 1$) для каждой вершины m
- И все?

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 ~= 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 исходящих 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

PageRank

Случайное блуждание по Web

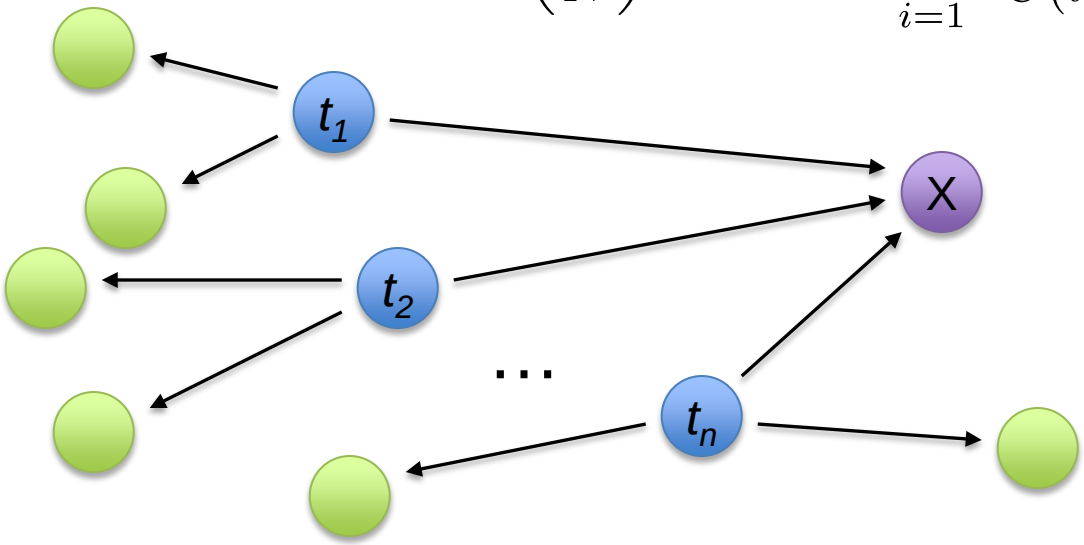
- Модель блуждающего веб-серфера
  - Пользователь начинает серфинг на случайной веб-странице
  - Пользователь произвольно кликает по ссылкам, тем самым перемещаясь от страницы к странице
- PageRank
  - Характеризует кол-во времени, которое пользователь провел на данной странице
  - Математически – это 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 $t_1 \ldots t_n$, где

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

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



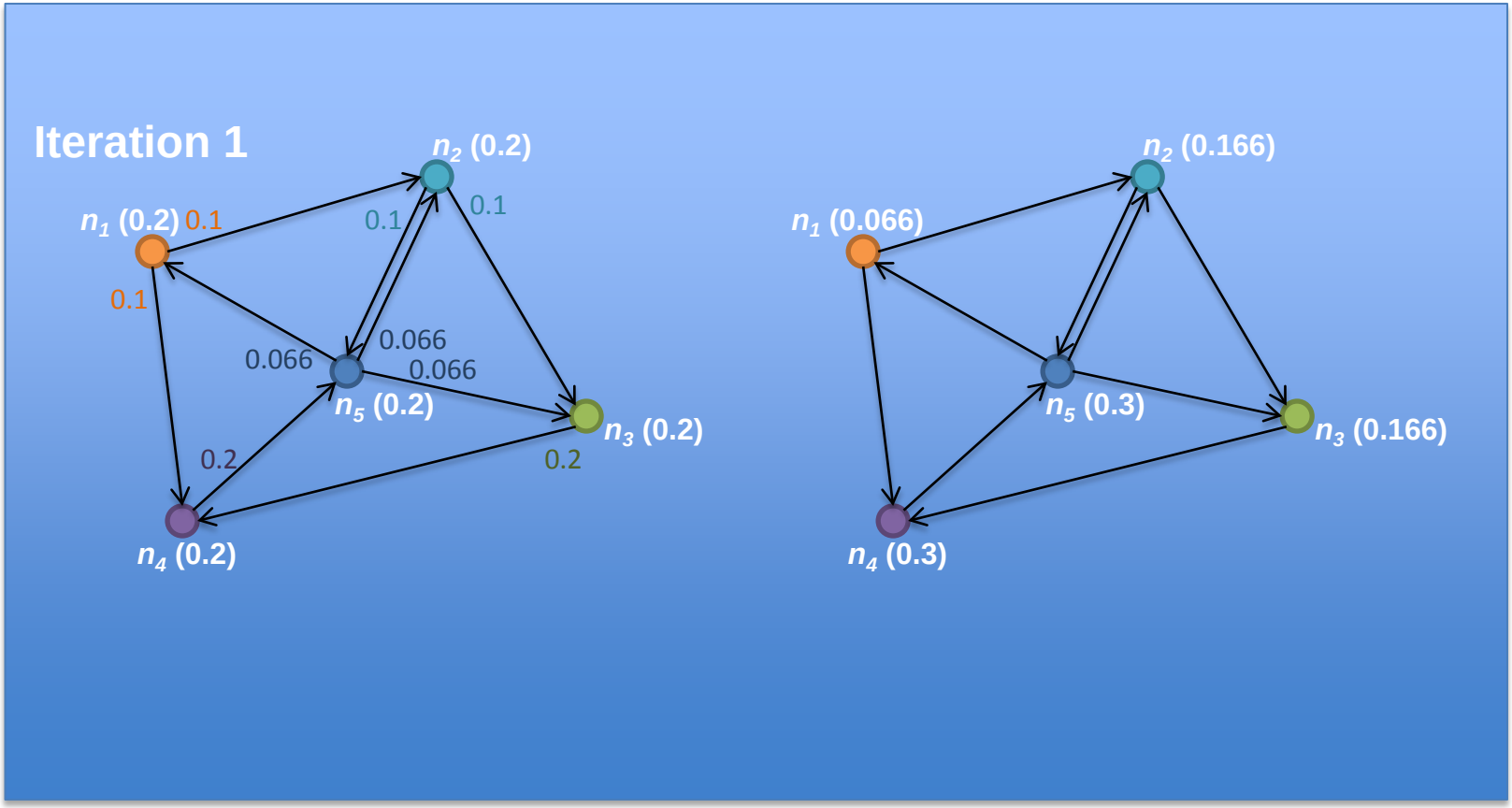
Вычисление PageRank

- Свойства PageRank'а
  - Может быть рассчитан итеративно
  - Локальный эффект на каждой итерации
- Набросок алгоритма
  - Начать с некоторыми заданными значения 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

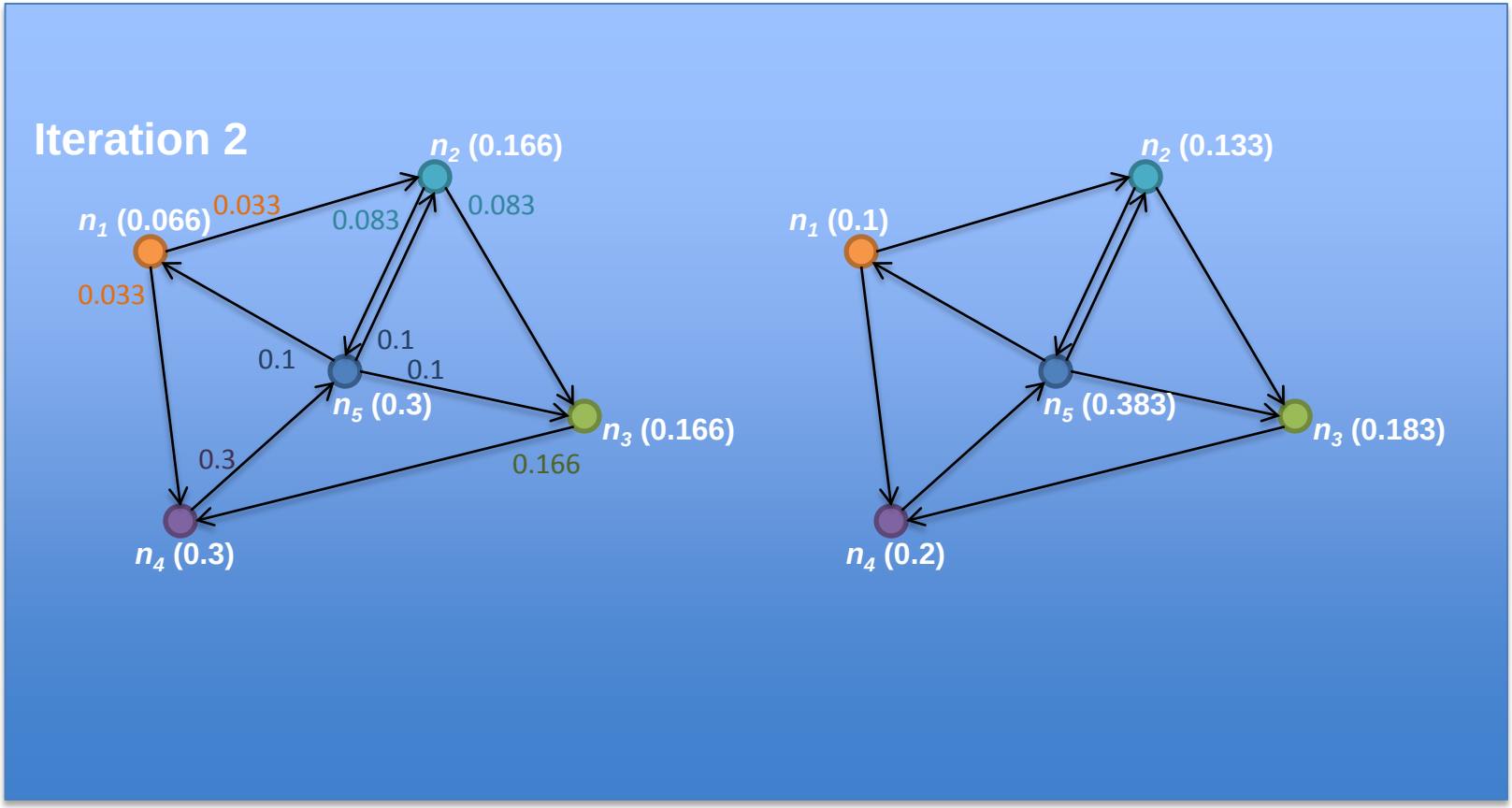
Упрощения для PageRank

- 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?

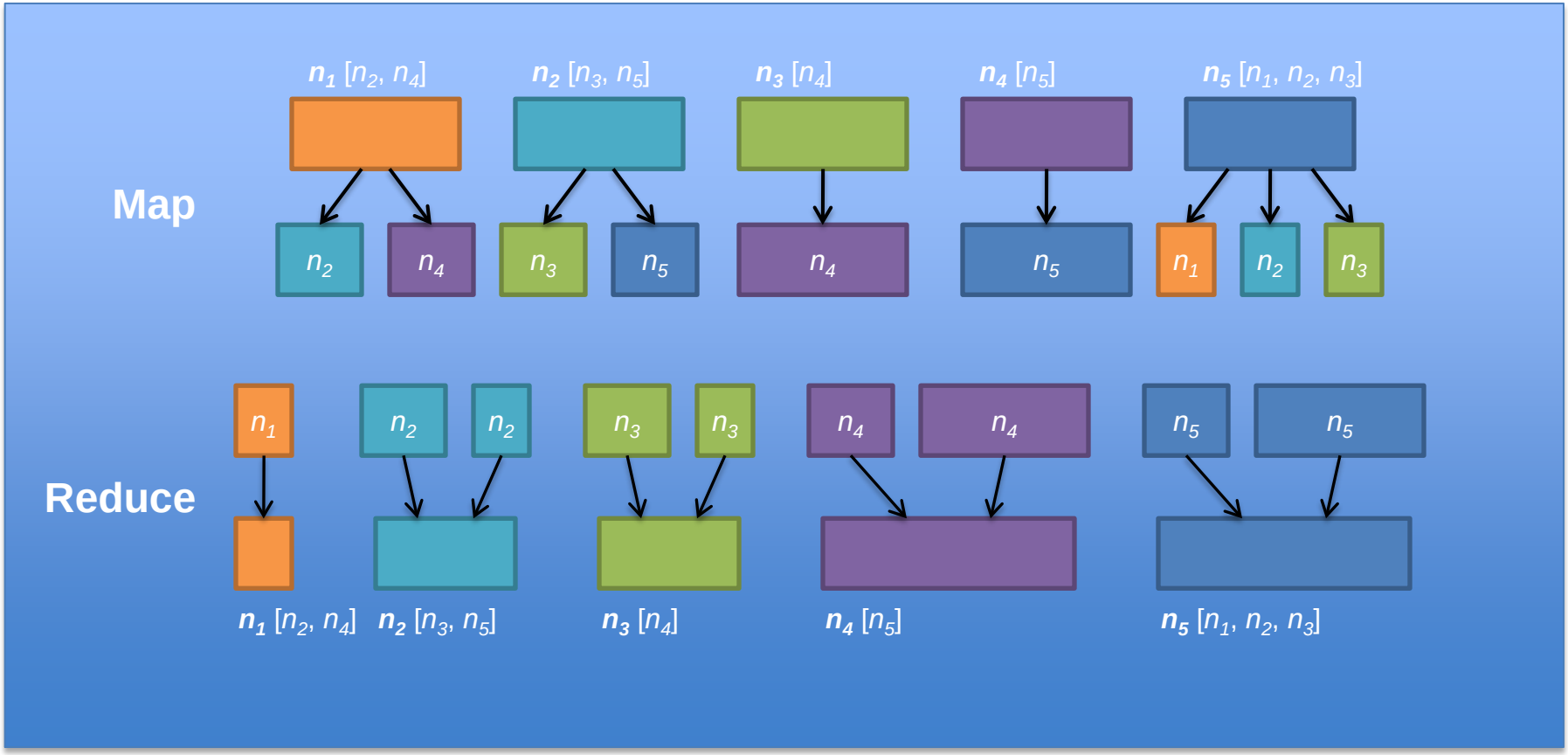
Пример расчета PageRank (1)



Пример расчета PageRank (2)



PageRank на MapReduce



PageRank: псевдокод

```
class Mapper
    method Map(nid n, node N)
        p ← N.PageRank/|N.AdjacencyList|
        Emit(nid n,N)                    // Pass along graph structure
        for all nodeid m ∈N.AdjacencyList do
            Emit(nid m, p)               // Pass PageRank mass to neighbors

class Reducer
    method Reduce(nid m, [p1, p2, . . .])
        M ← ∅
        for all p ∈counts [p1, p2, . . .] do
            if IsNode(p) then
                M ← p                    // Recover graph structure
            else
                s ← s + p                // Sum incoming PageRank contributions
        M.PageRank← 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
- Сходимость для Web-графа?
  - Это так такой простой вопрос
- Аккуратней со ссылочным спамом:
  - Ссылочные фермы
  - Ловушки для краулеров (*Spider traps)*
  - …

Кроме PageRank…

- Вариации PageRank
  - Взвешенные ребра
  - Персонализированный PageRank
- Вариации на тему “graph random walks”
  - Hubs и authorities (HITS)
  - SALSA

Приложение PageRank

- Статическая приоритезация для веб-ранжирования
- Определение “особенных 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 статистки
  - Коэффициенты кластеризации
  - Подсчет треугольников

Основные проблемы для алгоритмов на графах

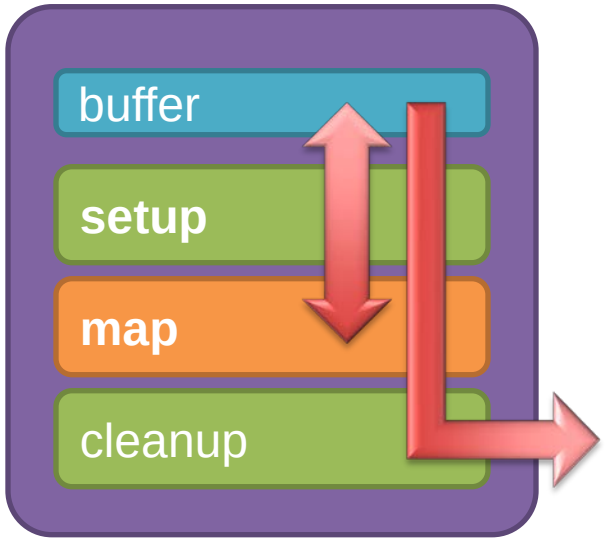
- Большие разреженные графы
- Топология графов

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, аггрегировать сообщения в буфер, писать содержимое буфера в конце
  - Минус: требуется управление памятью



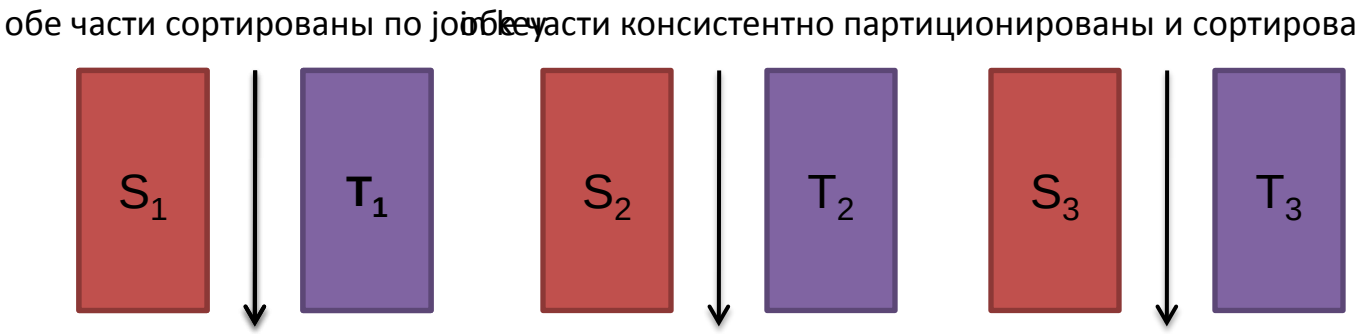
Emit all key-value pairs at once

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

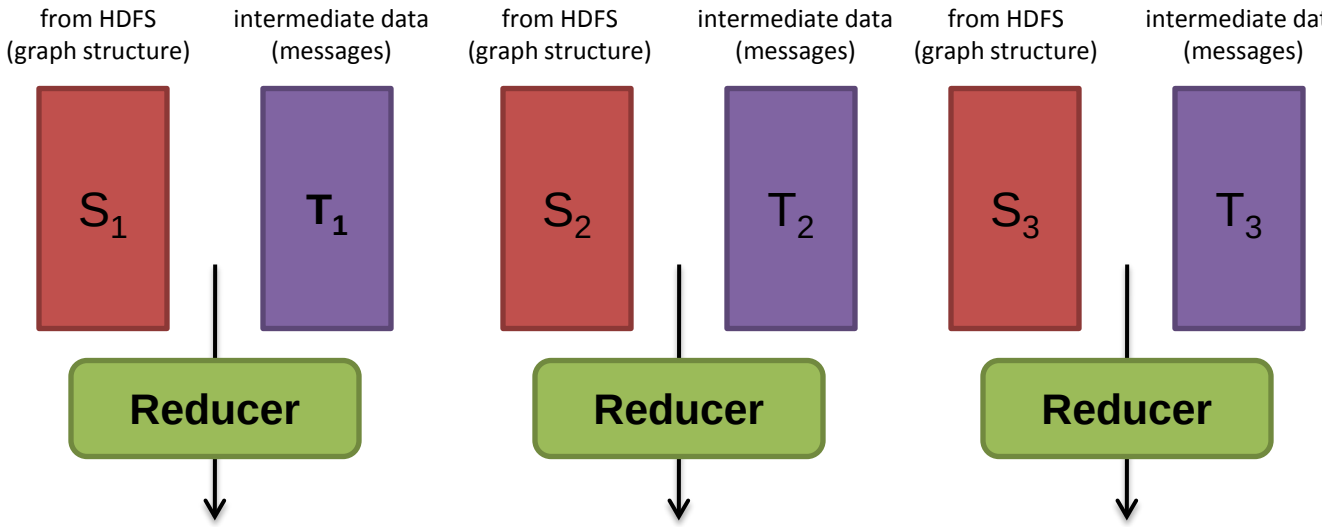
Schimmy Design Pattern

- 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*



Используем Schimmy

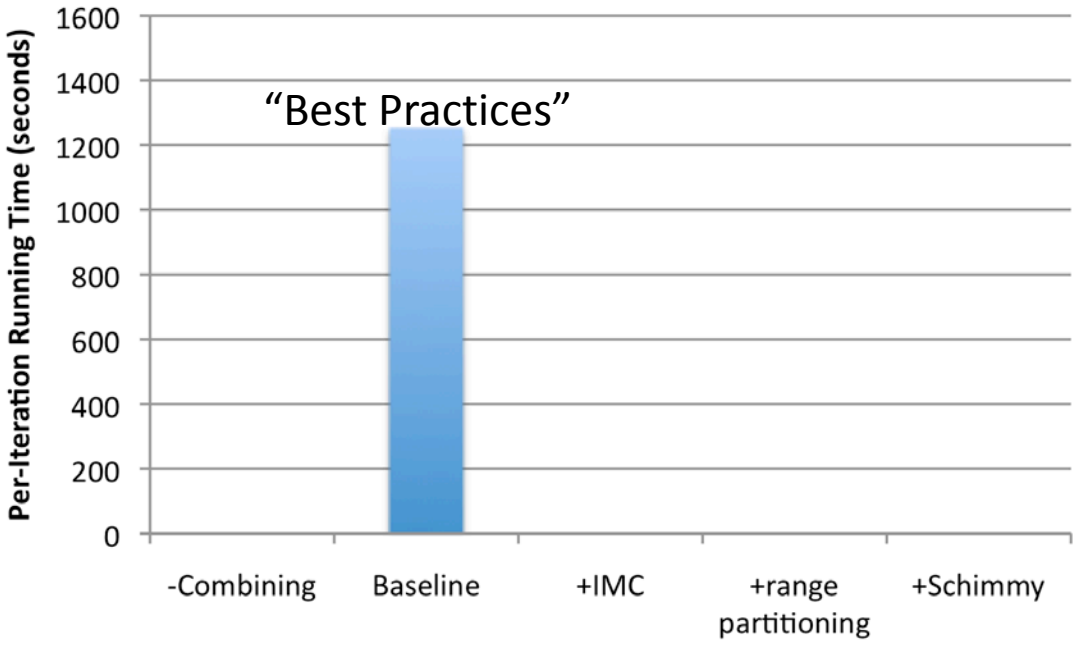
- Schimmy = на редьюсерах выполняется параллельный *merge join* между *graph structure* и *messages*
  - Консистентное партиционирование между входным и промежуточными данными (*intermediate data*)
  - *Mappers* пишут только *messages* (актуальные вычисления)
  - *Reducers* читают *graph structure* напрямую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



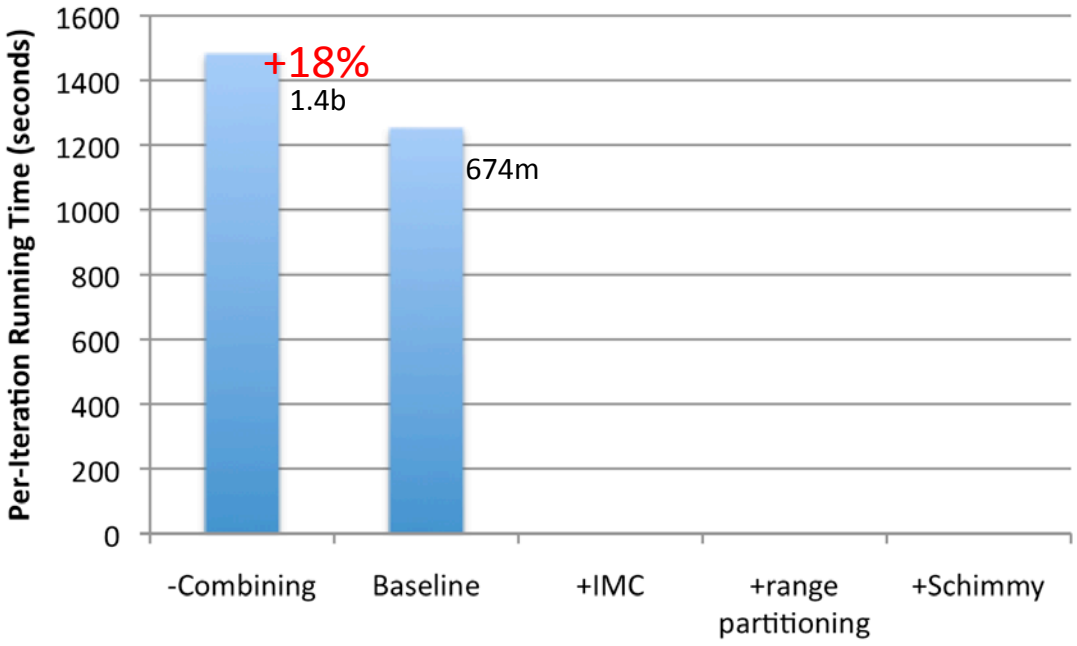
Эксперимент

- 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

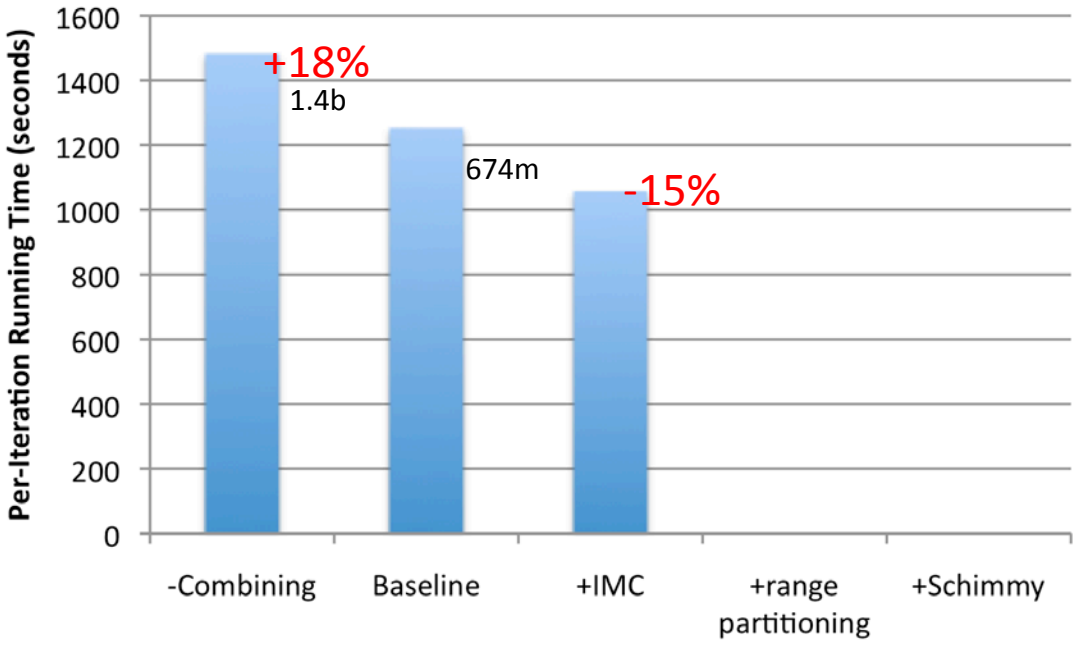
Результаты


Per-Iteration Running Time (seconds)
1600
1400
1200
1000
800
600
400
200
0
"Best Practices"
-Combining
Baseline
+IMC
+range partitioning
+Schimmy

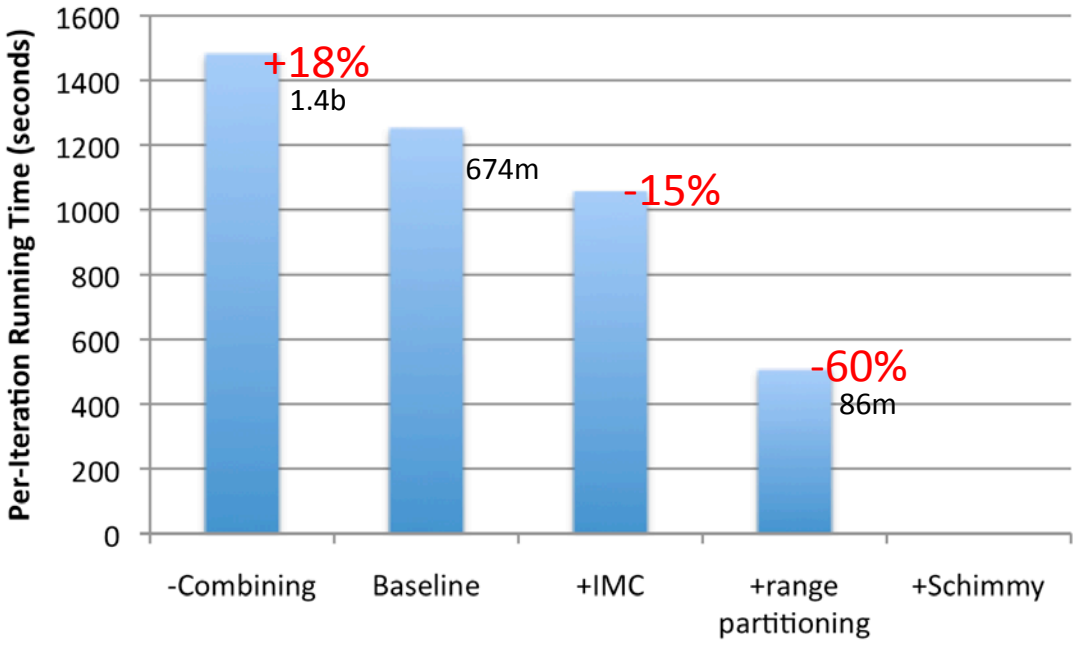
Результаты


Per-Iteration Running Time (seconds)
1600
1400
1200
1000
800
600
400
200
0
+18%
1.4b
674m
-Combining
Baseline
+IMC
+range partitioning
+Schimmy

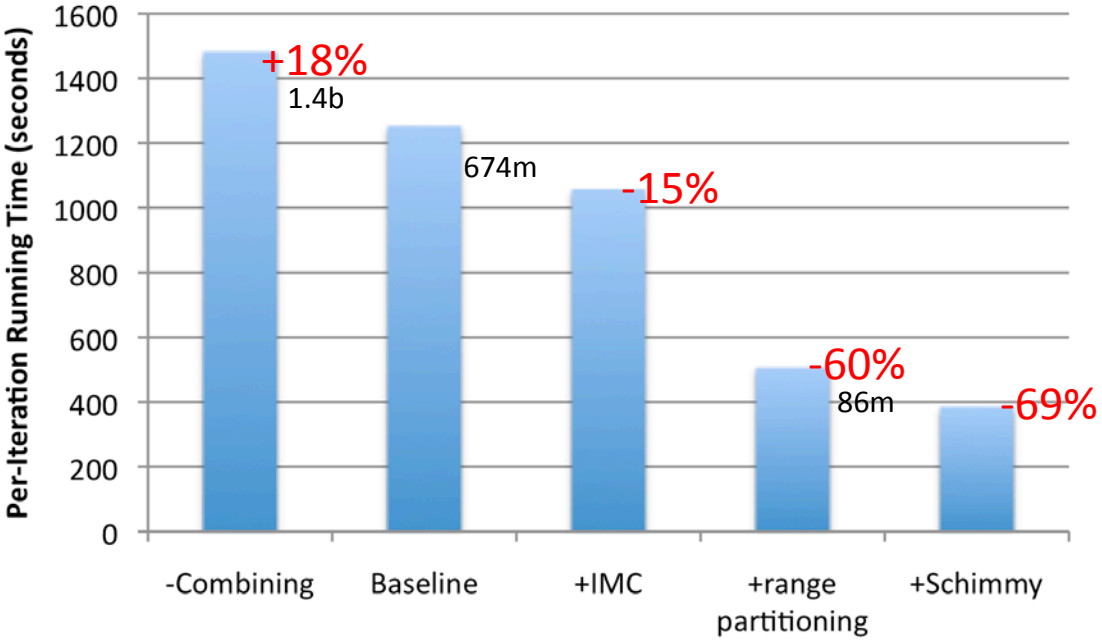
Результаты


Per-Iteration Running Time (seconds)
1600
1400
1200
1000
800
600
400
200
0
+18%
1.4b
674m
-15%
-Combining
Baseline
+IMC
+range partitioning
+Schimmy

Результаты


Per-Iteration Running Time (seconds)
1600
1400
1200
1000
800
600
400
200
0
+18%
1.4b
674m
-15%
-60%
86m
-Combining
Baseline
+IMC
+range partitioning
+Schimmy

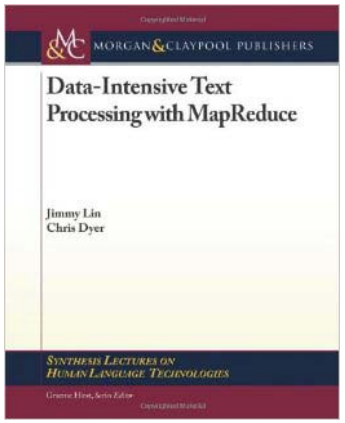
Результаты


Per-Iteration Running Time (seconds)
1600
1400
1200
1000
800
600
400
200
0
+18%
1.4b
674m
-15%
-60%
86m
-69%
-Combining
Baseline
+IMC
+range partitioning
+Schimmy

Ресурсы

Data-Intensive Text Processing with MapReduce
Jimmy Lin and Chris Dyer (Authors) (April, 2010)

Chapter5: Graph Algorithms



Вопросы?