ТЕХНОСФЕРА

Лекция №6

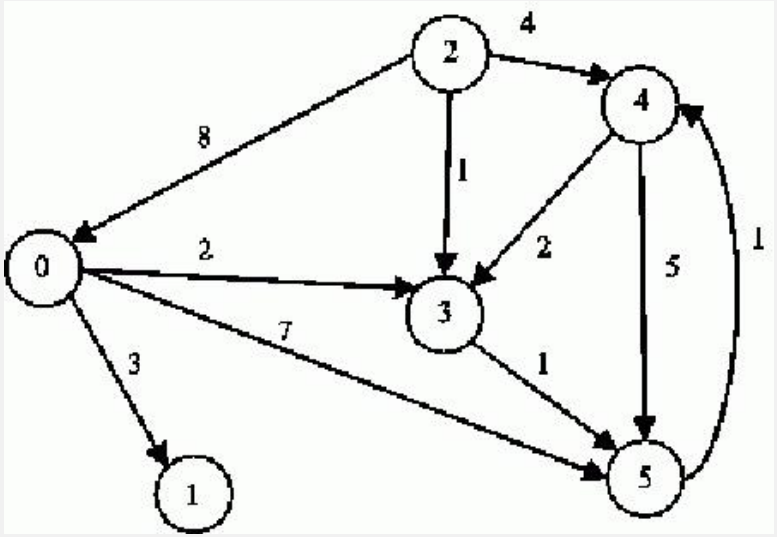
MapReduce в Hadoop
Графы

Евгений Чернов

Граф как структура данных

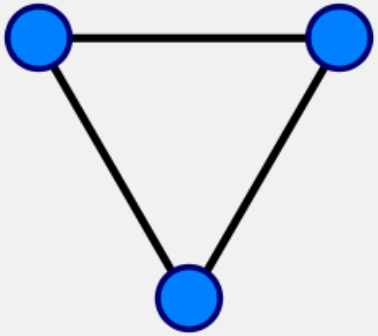
G = (V,E), где

- V представляет собой множество вершин (nodes)
- E представляет собой множество ребер (edges/links)
- Ребра и вершины могут содержать дополнительную информацию

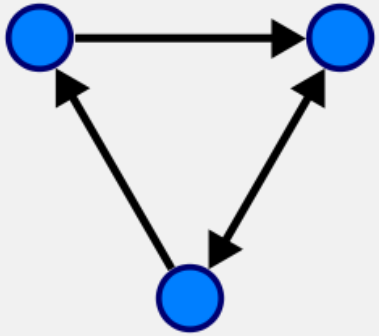


- $V = \{0, 1, 2, 3, 4, 5\}$
- $E = \{[0,1], [0,3], [0,5], \ldots\}$
- $W_{0,1} = 3$, $W_{0,3} = 2$, …

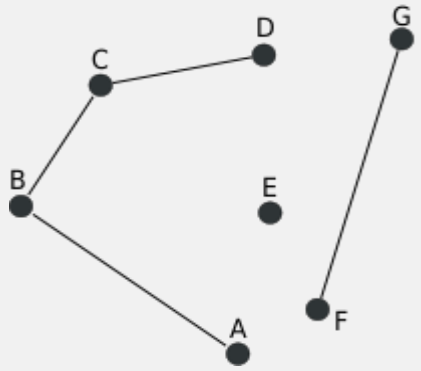
Примеры графов



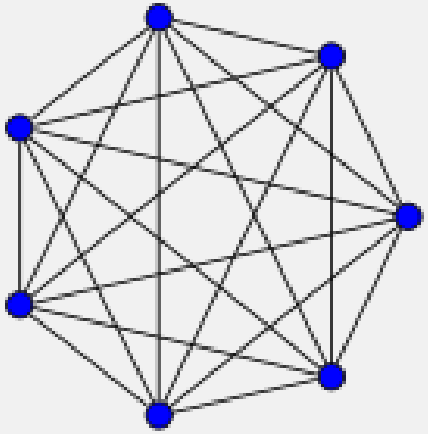
Неориентированный



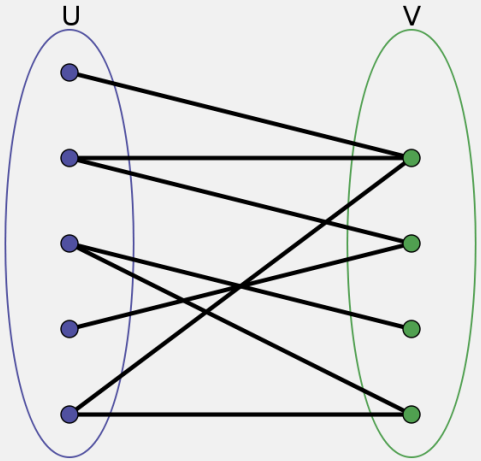
Ориентированный



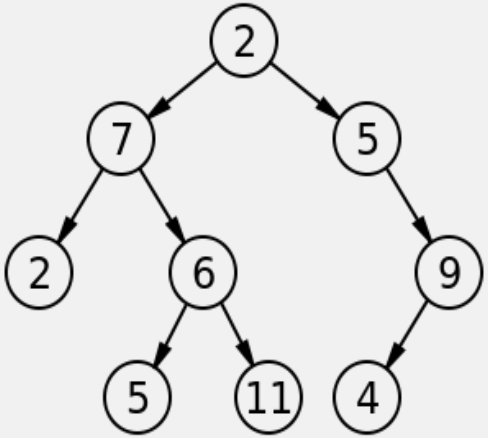
Несвязный



Полный

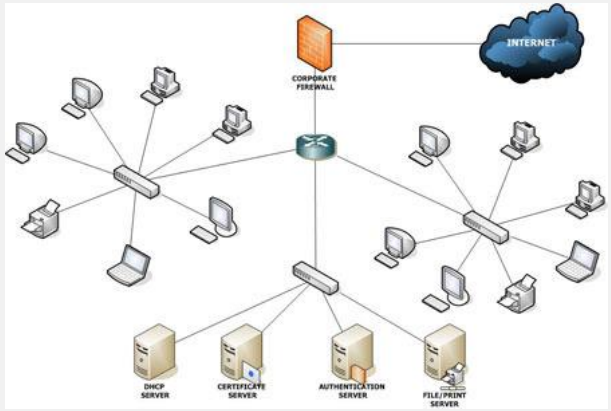


Двудольный

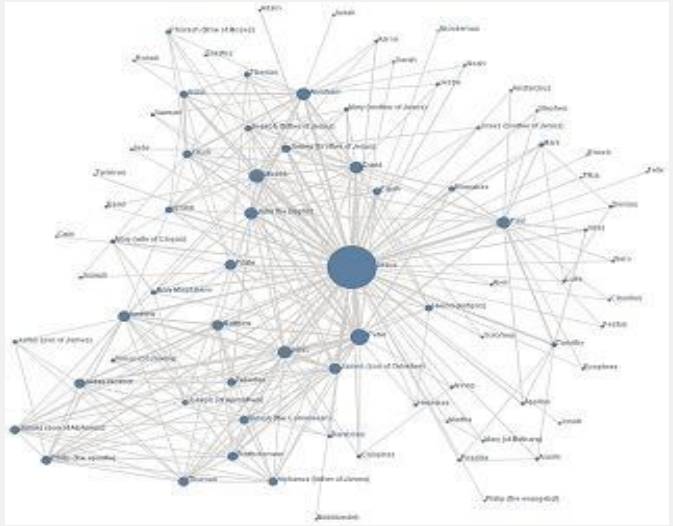


Дерево

Графы есть практически везде

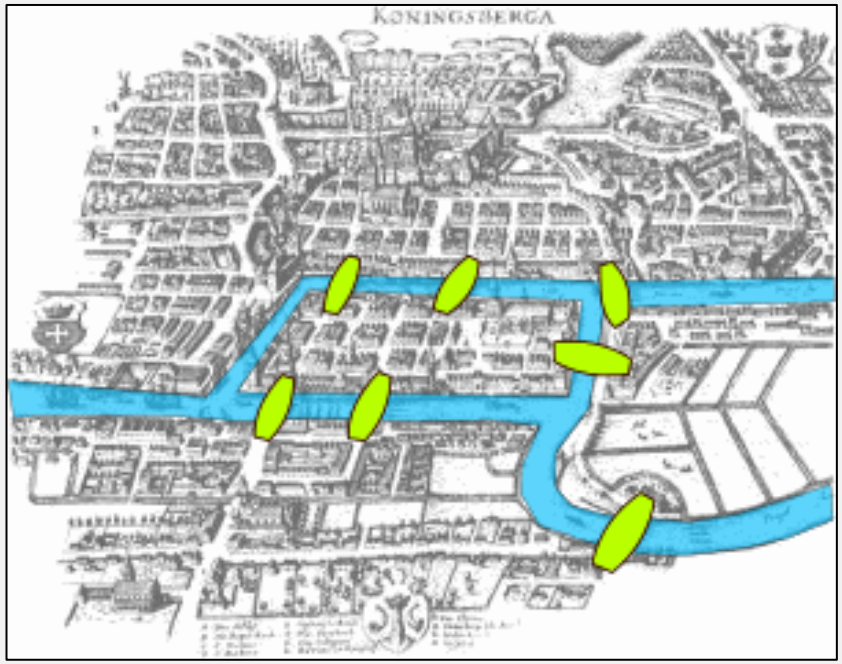


- Структура компьютеров и серверов сети
- Сайты/страницы и ссылки в Web
- Социальные сети
- Структура дорог/жд/метро и т.д.



Задачи и проблемы на графах

- **Поиск кратчайшего пути**
  - Роутинг траффика
  - Навигация маршрута
- **Поиск минимального остовного дерева (Minimum Spanning Tree)**
  - Телекоммуникационные компании
- **Поиск максимального потока (Max Flow)**
  - Структура компьютеров и серверов Интернет
- **Поиск паросочетаний (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 вершине
- Обход графа

Ключевые вопросы

- Как представить граф в MapReduce?
- Как обходить граф в MapReduce?

Представление графов

G = (V, E)

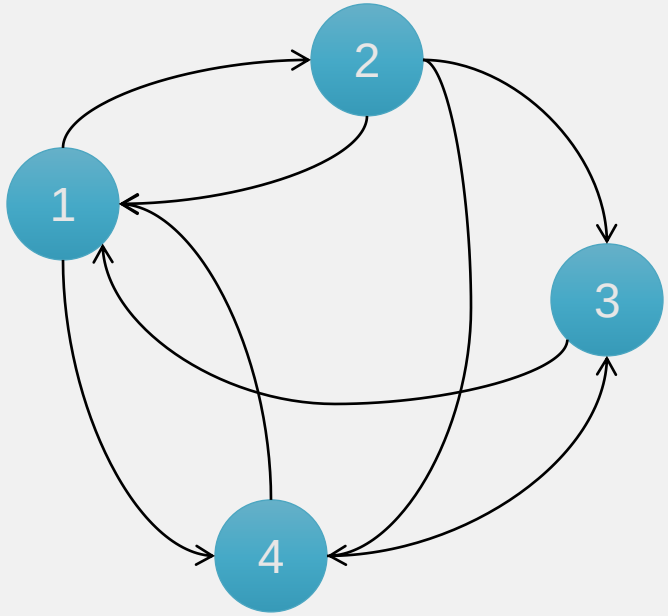
- Матрица смежности
- Списки смежности

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 соответствует переходу по входящим и исходящим ссылкам

Минусы

- Матрица разреженная, множество лишних нулей
- Расходуется много лишнего места

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

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 вершины

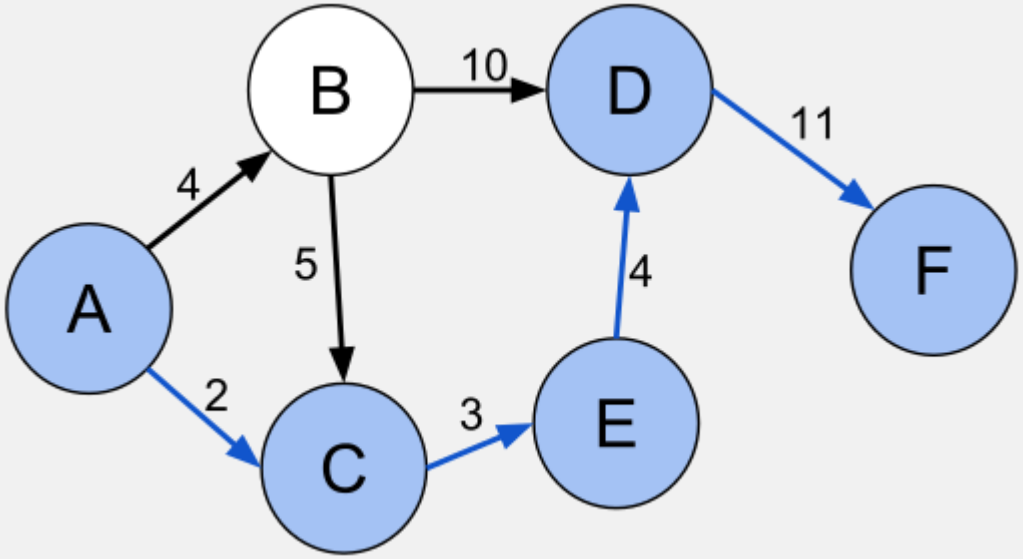
Минусы

- Намного сложнее подсчитать входящие ссылки

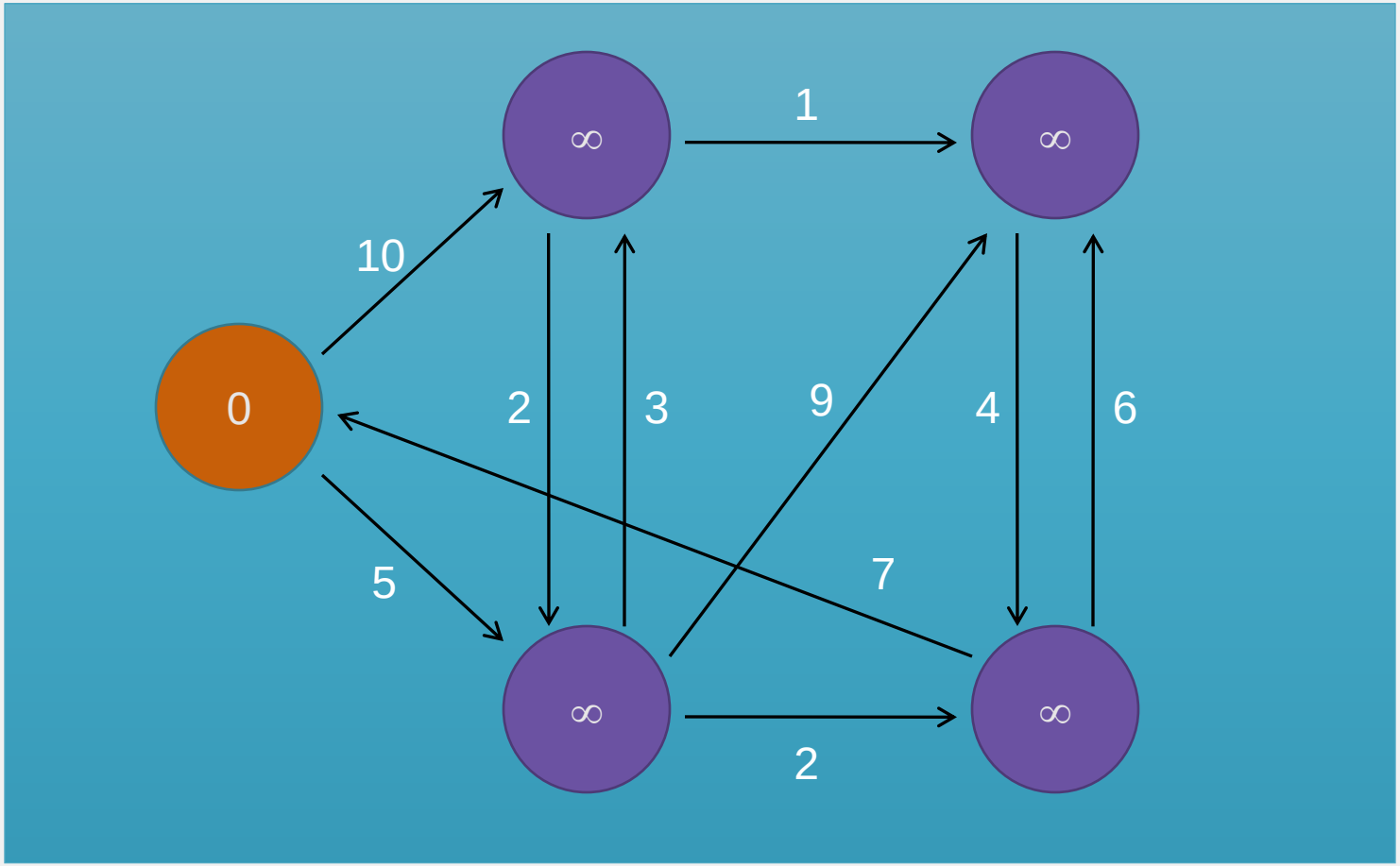
Задача

Поиск кратчайшего пути в графе

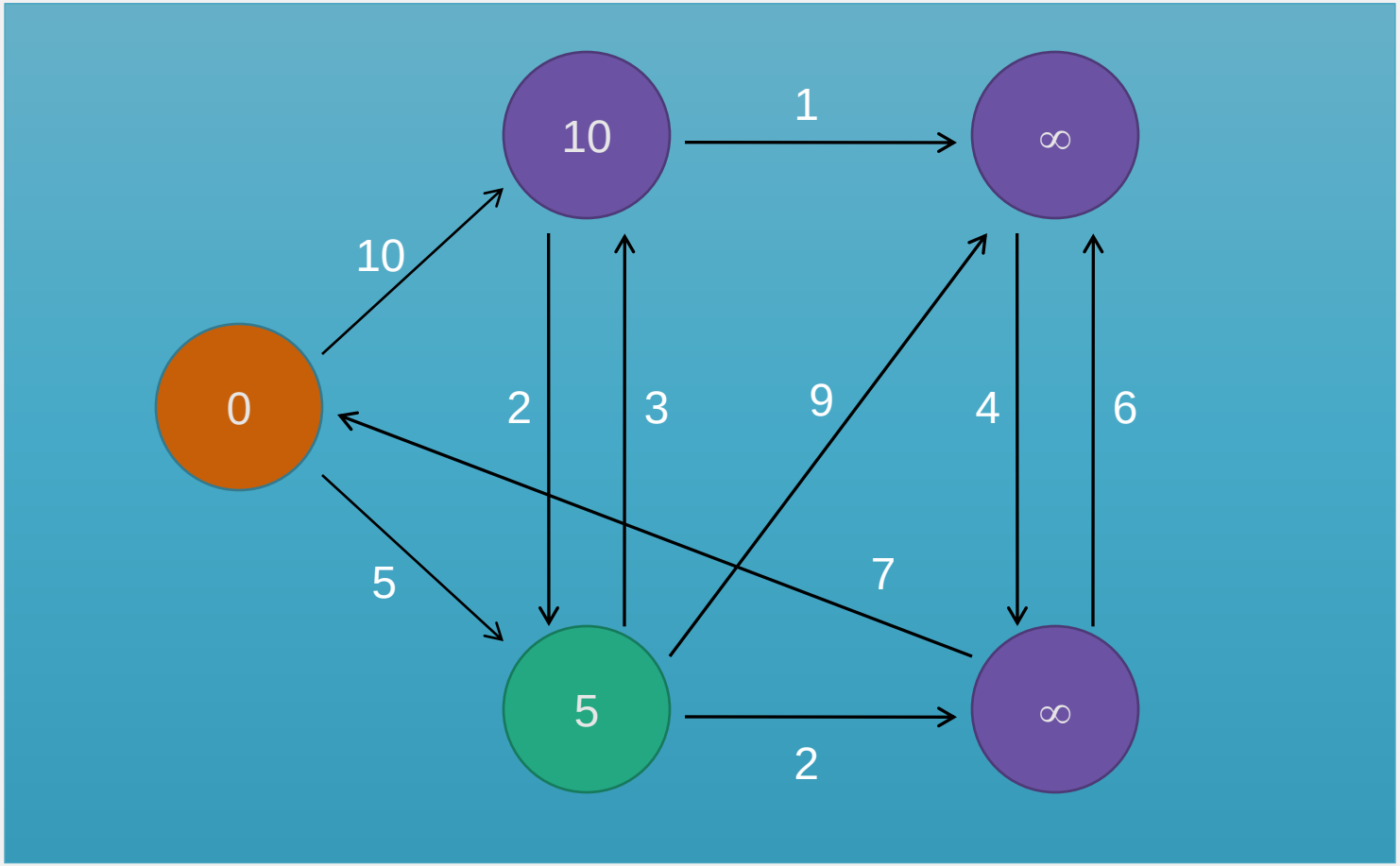
- Найти кратчайший путь от исходной вершины до заданной (или до нескольких заданных)
- Также, кратчайший может означать с наименьшим общем весом всех ребер



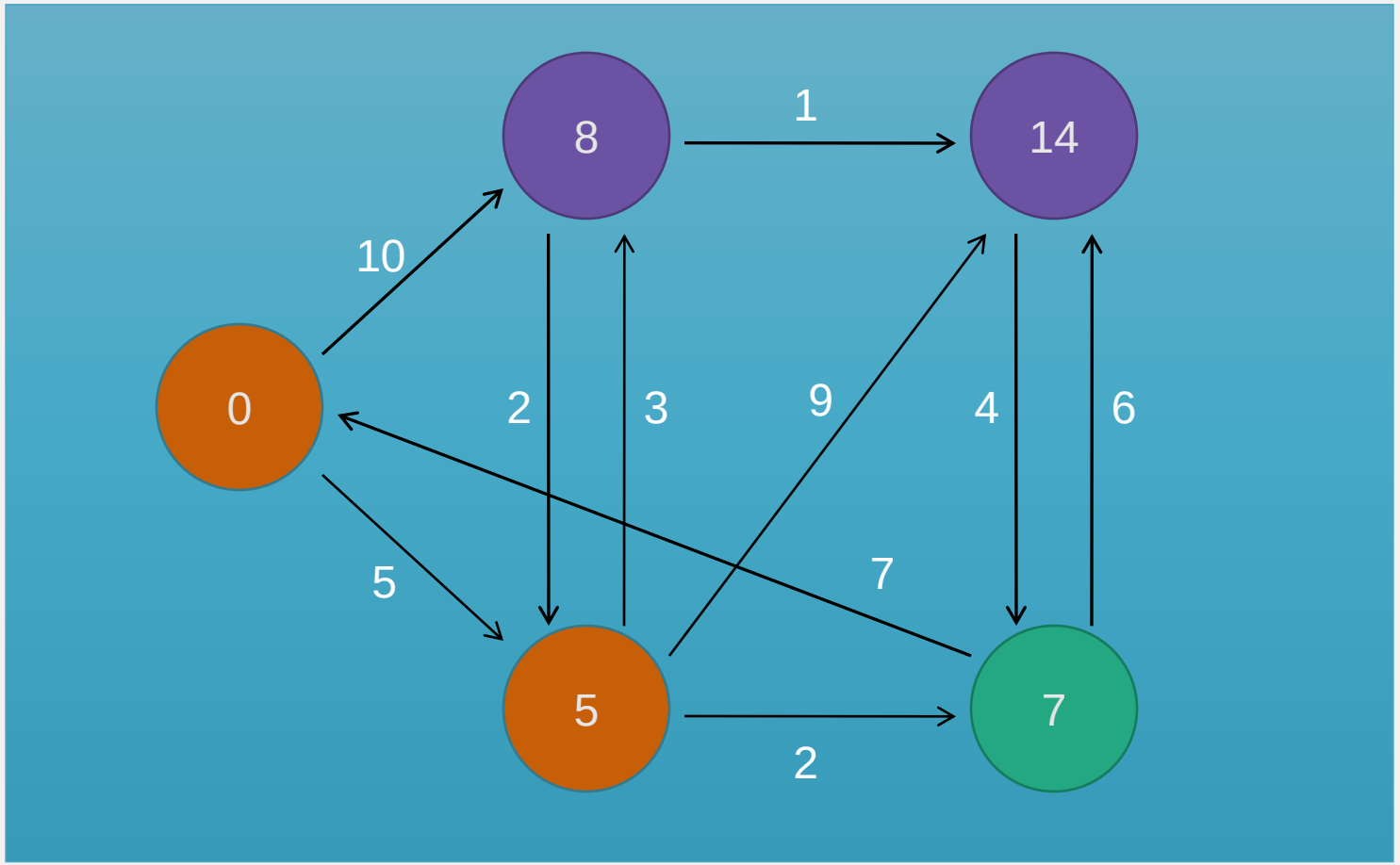
Алгоритм Дейкстры (пример)



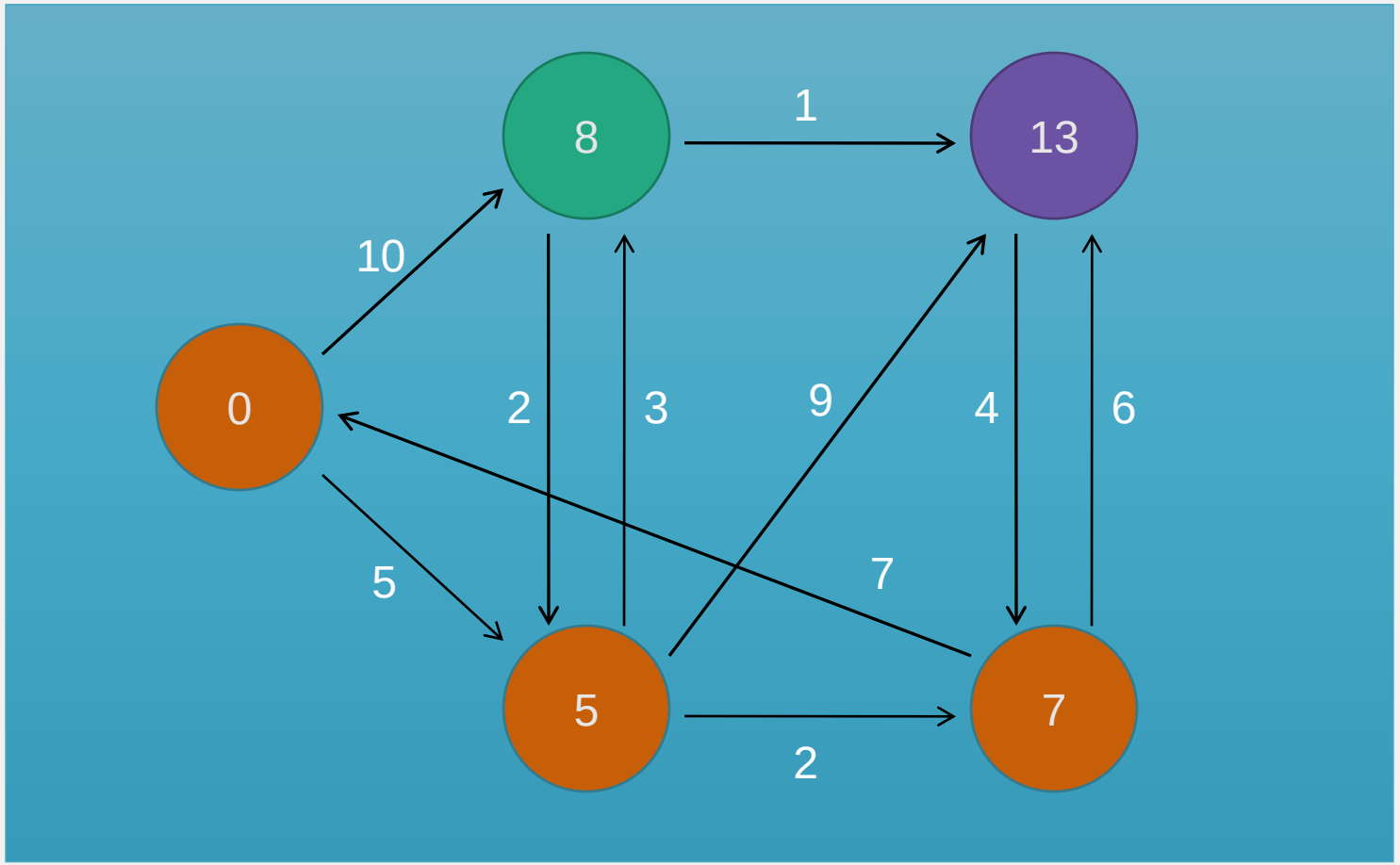
Алгоритм Дейкстры (пример)



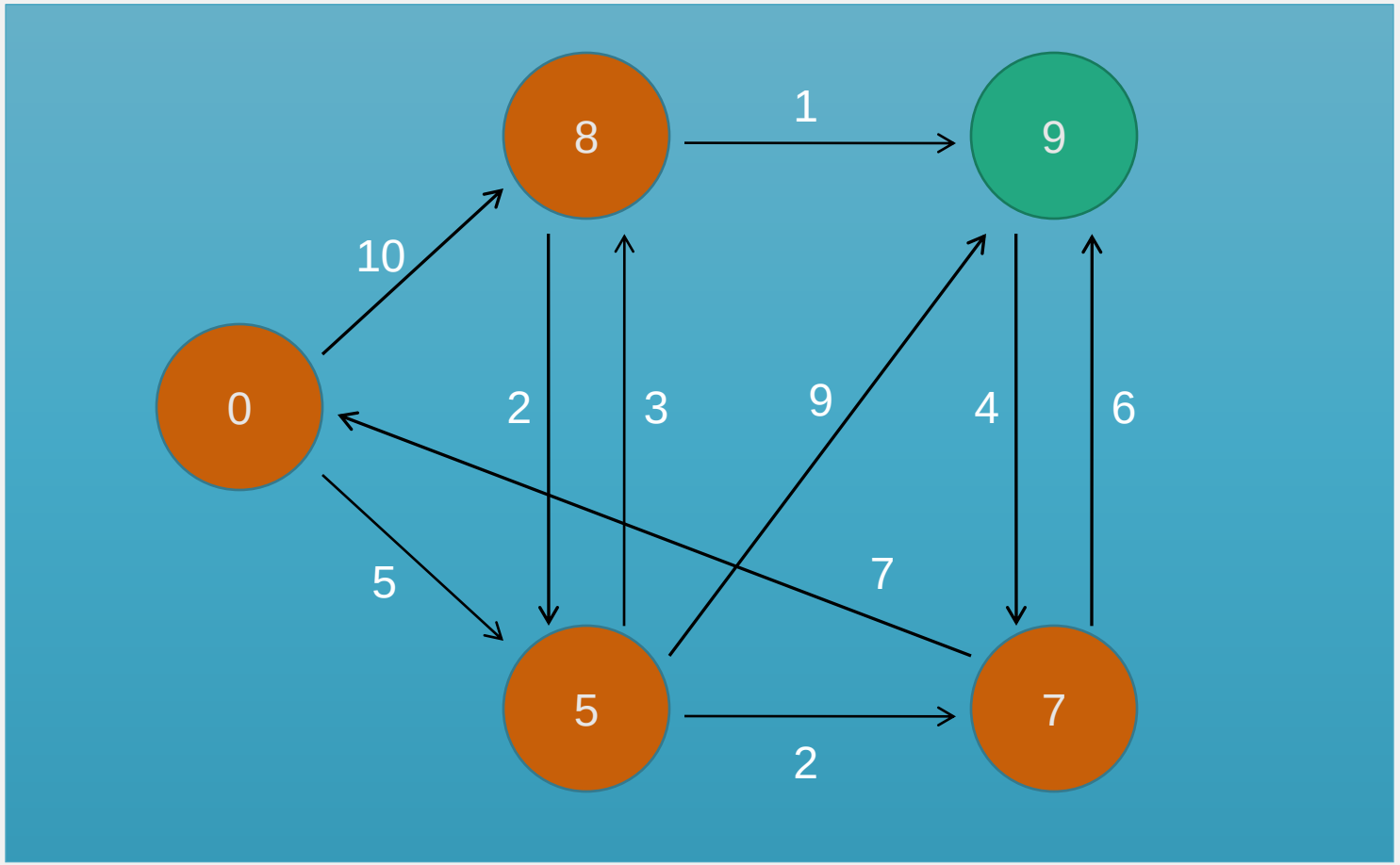
Алгоритм Дейкстры (пример)



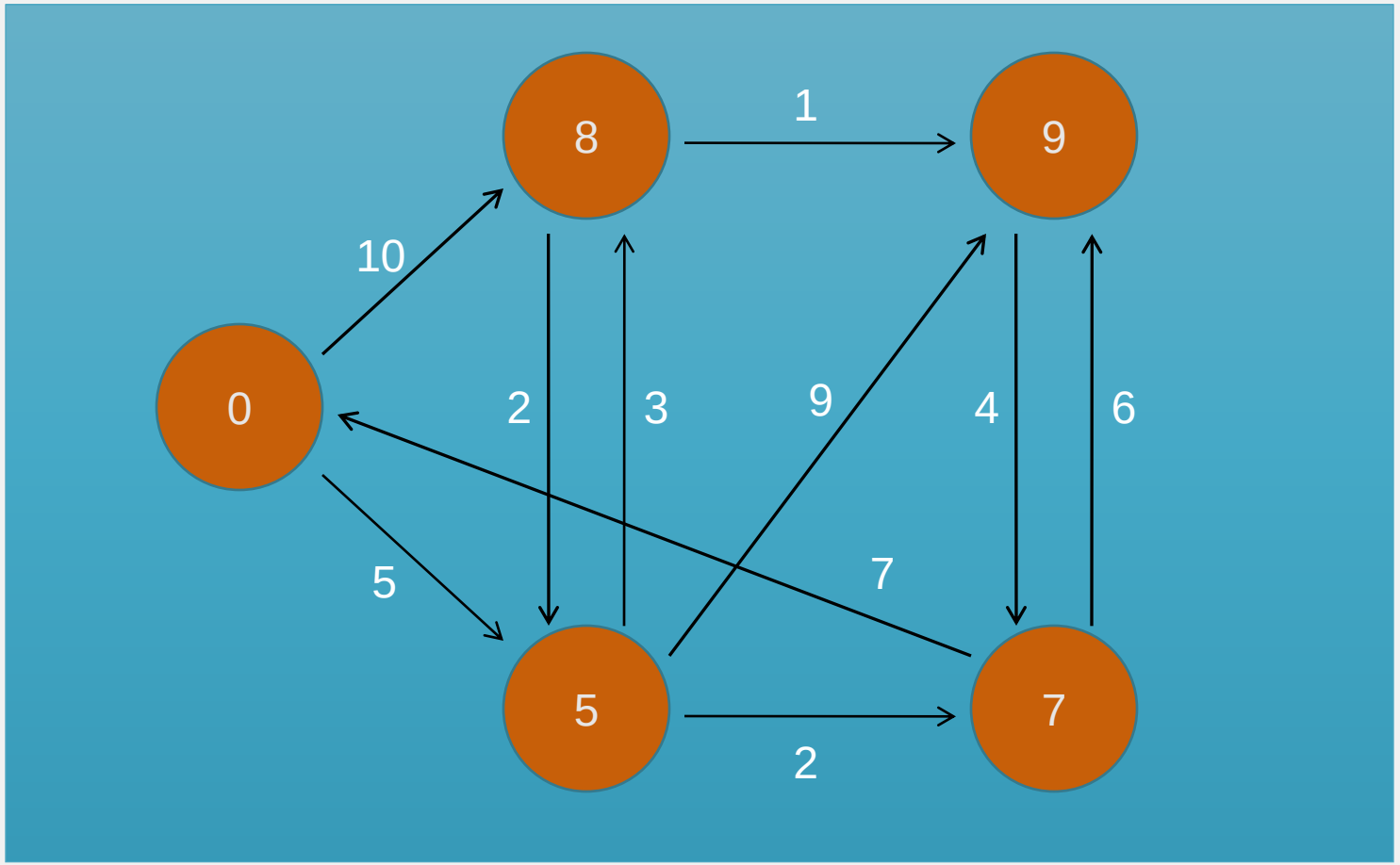
Алгоритм Дейкстры (пример)



Алгоритм Дейкстры (пример)



Алгоритм Дейкстры (пример)

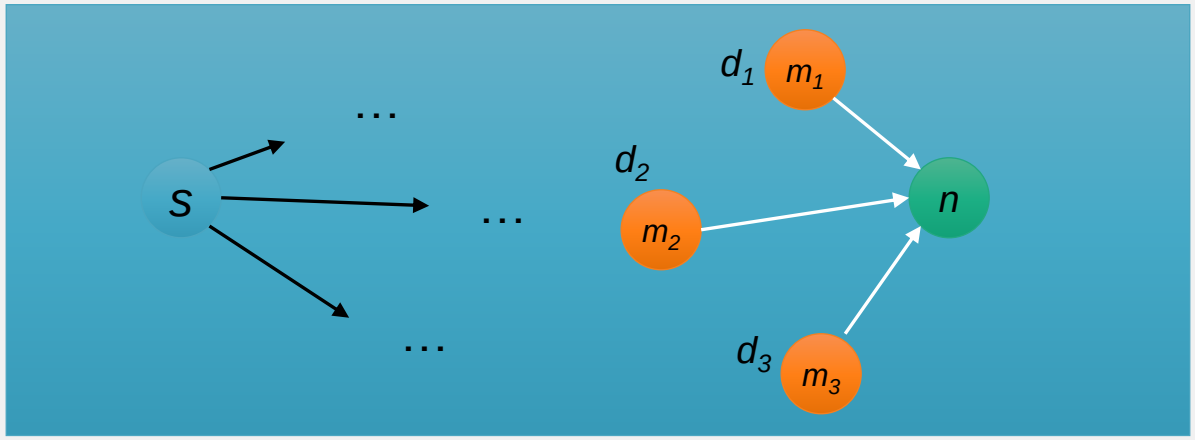


Алгоритм Дейкстры

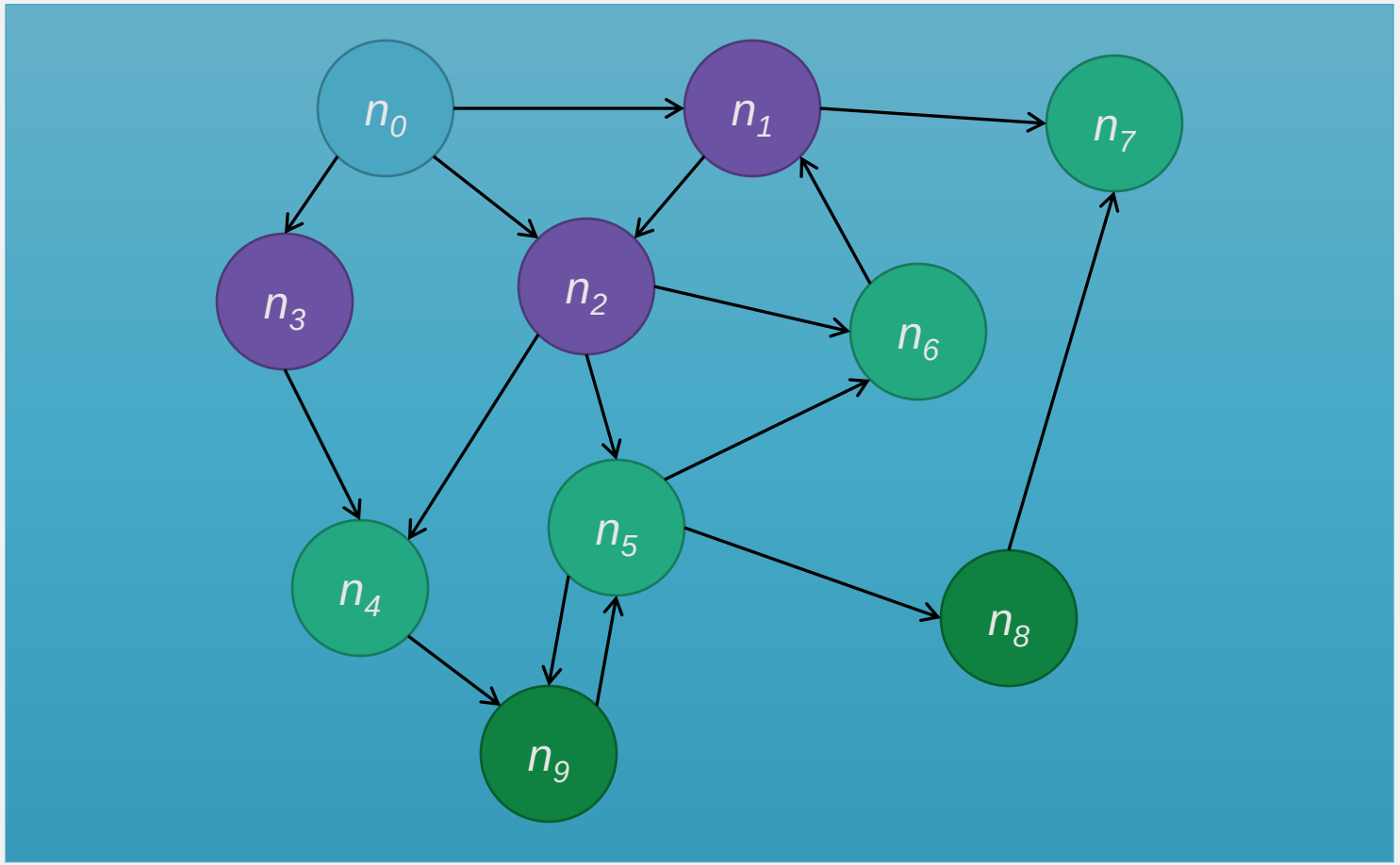
```
1.  Dijkstra(V, s, w)
2.      for all vertex v ∈ V do
3.              d[v] ← ∞
4.      d[s] ← 0
5.      Q ← {V }
6.      while Q != ∅ do
7.              u ←ExtractMin(Q)
8.              for all vertex v ∈ u.AdjacencyList do
9.                      if d[v] > d[u] + w(u, v) then
10.                             d[v] ← d[u] + w(u, v)
```

Поиск кратчайшего пути

- Рассмотрим простой случай, когда вес всех ребер одинаков (равен 1)
- Решение проблемы можно решить по индукции:
  - Расстояние до источника равно 0:
    DistanceTo(s) = 0
  - Для всех вершин *p*, достижимых из s, растояние равно 1:
    DistanceTo(p) = 1
  - Для всех вершин n, достижимых из других множеств *M*
    DistanceTo(n) = 1 + min(DistanceTo(m), m ∈ M)

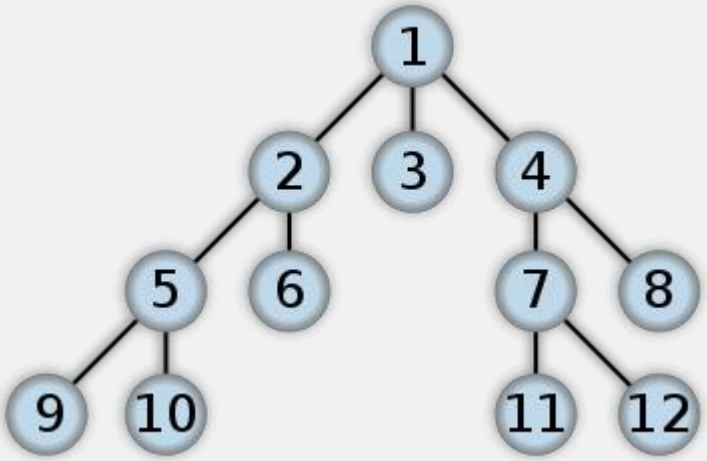


Параллельный поиск в ширину (BFS)



Breadth First Search: представление данных

- Key: вершина *n*
- Value: *d* (расстояние от начала), adjacency list (вершины, доступные из *n*)
- Инициализация: для всех вершин, кроме начальной, *d* = ∞




```
1 -> [0, {2, 3, 4}]
2 -> [∞, {5, 6} ]
3 -> [∞, {} ]
4 -> [∞, {7, 8} ]
5 -> [∞, {9, 10} ]
…
```

Breadth First Search: Mapper

mapper(key, value):

emit(key, value)

$\forall m \in$ value.adjacency_list: emit (m, value.d + 1)

Mapper 1

```
1 -> [0, {2, 3, 4}]
```

```
1 -> [0, {2, 3, 4}]
2 -> [1, {} ]
3 -> [1, {} ]
4 -> [1, {} ]
```

Mapper 2

```
2 -> [∞, {5, 6} ]
```

```
2 -> [∞, {5, 6}]
5 -> [∞, {} ]
6 -> [∞, {} ]
```

Breadth First Search: Reducer

- Sort/Shuffle
  - Сгруппировать расстояния по достижимым вершинам
- Reducer:
  - Выбрать путь с минимальным расстоянием для каждой достижимой вершины
  - Сохранить структуру графа

Reduce In:

```
2 -> {[1, {} ], [∞, {5, 6}]}
```

Reduce Out:

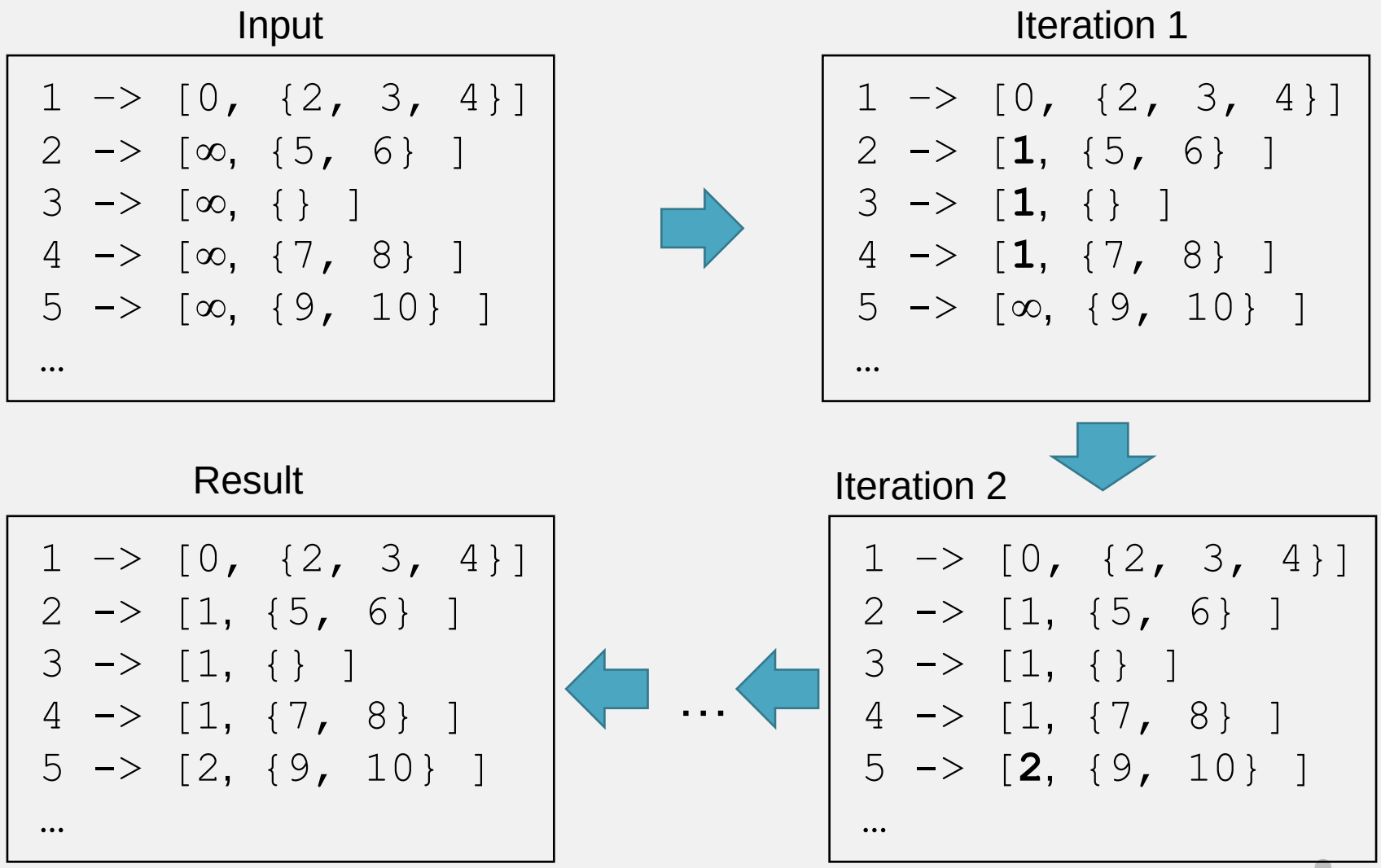
```
2 -> [1, {5, 6} ]
```

BFS: псевдокод

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


1.  class Reducer
2.      method Reduce(nid m, [d1, d2, . . .])
3.          dmin ← ∞
4.          M ← ∅
5.          for all d ∈ counts [d1, d2, . . .] do
6.              if IsNode(d) then
7.                  M ← d          // Recover graph structure
8.              else if d < dmin then
9.                  dmin ← d
10.         M.Distance ← dmin      // Update shortest distance
11.         Emit(nid m, node M)
```

Breadth First Search: Итерации



Breadth First Search: 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: критерий завершения

- Как много итераций нужно для завершения параллельного BFS?
- Когда первый раз посетили искомую вершину, значит найден самый короткий путь
- 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?

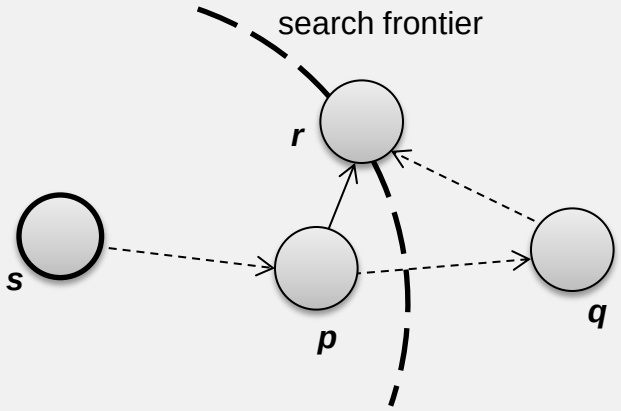
BFS: Weighted Edges

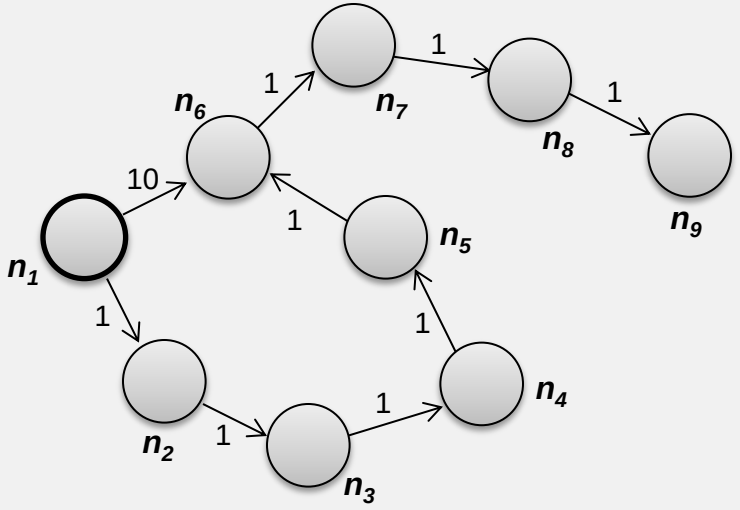
- Добавим положительный вес каждому ребру
  - Почему вес ребра не может быть отрицательным?
- Простая доработка: добавим вес *w* для каждого ребра в список смежных вершин
  - В mapper, emit (m, $d + w_p$) вместо (m, $d + 1$) для каждой вершины m
- И все?

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

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 данным из входящих вершин
- Повторять итерации до выполнения критерия сходимости, который контролируется внешним драйвером
- Передавать структуру графа между итерациями

PageRank: Случайное блуждание по Web

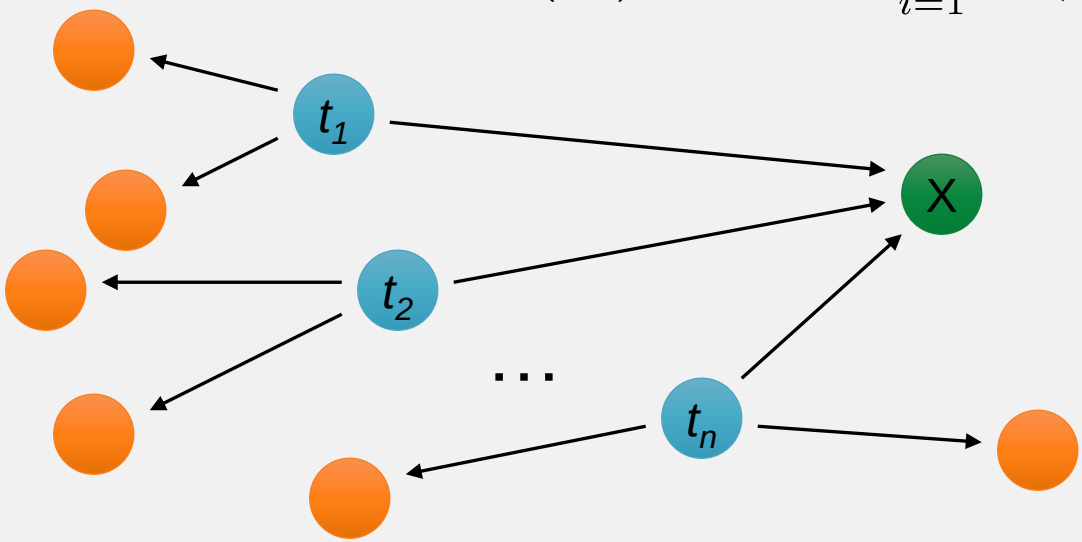
- Модель блуждающего веб-серфера
  - Пользователь начинает серфинг на случайной веб-странице
  - Пользователь произвольно кликает по ссылкам, тем самым перемещаясь от страницы к странице
- PageRank
  - Характеризует кол-во времени, которое пользователь провел на данной странице
  - Математически – это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

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



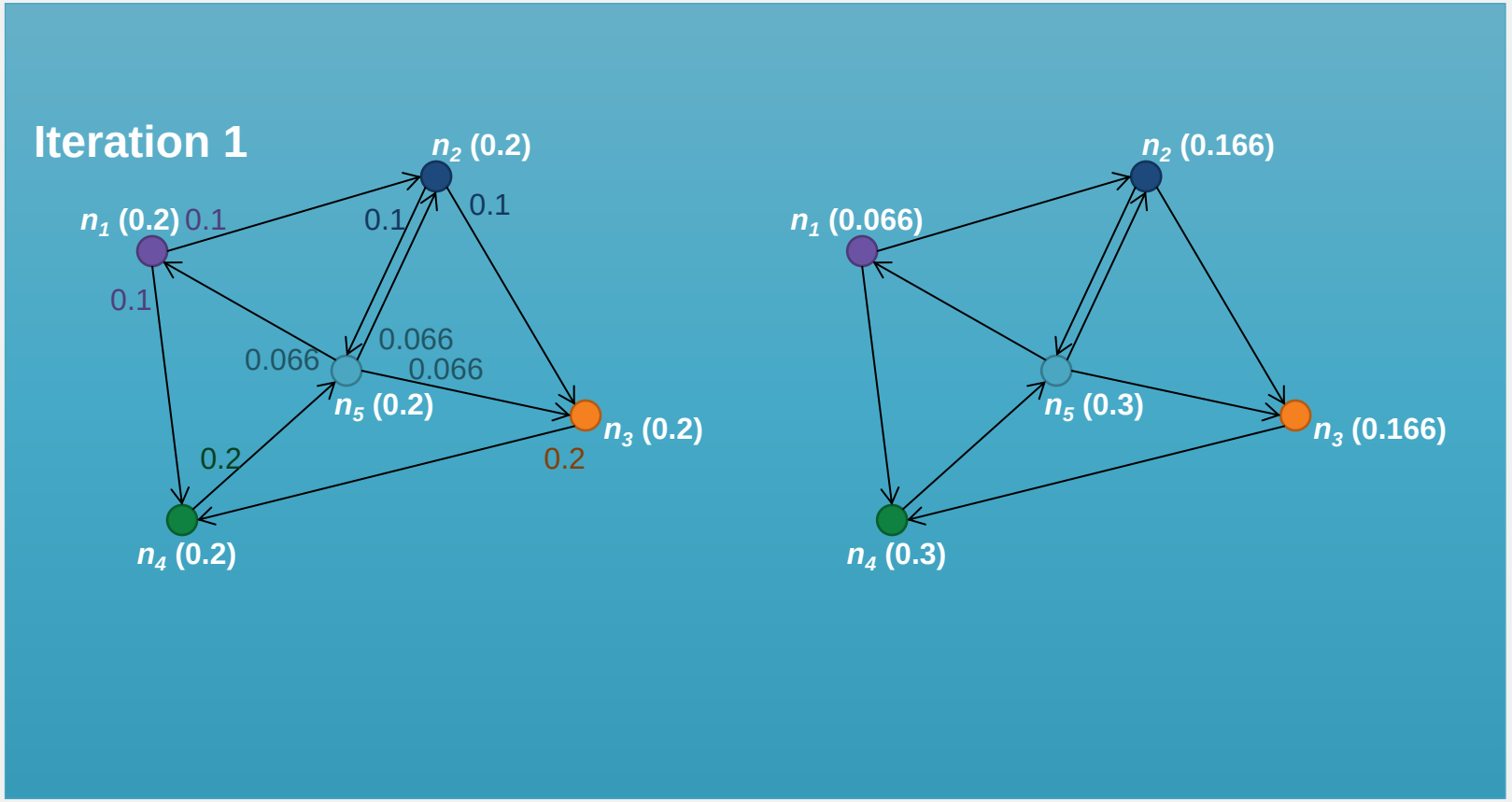
Вычисление PageRank

- Свойства PageRank’a
  - 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

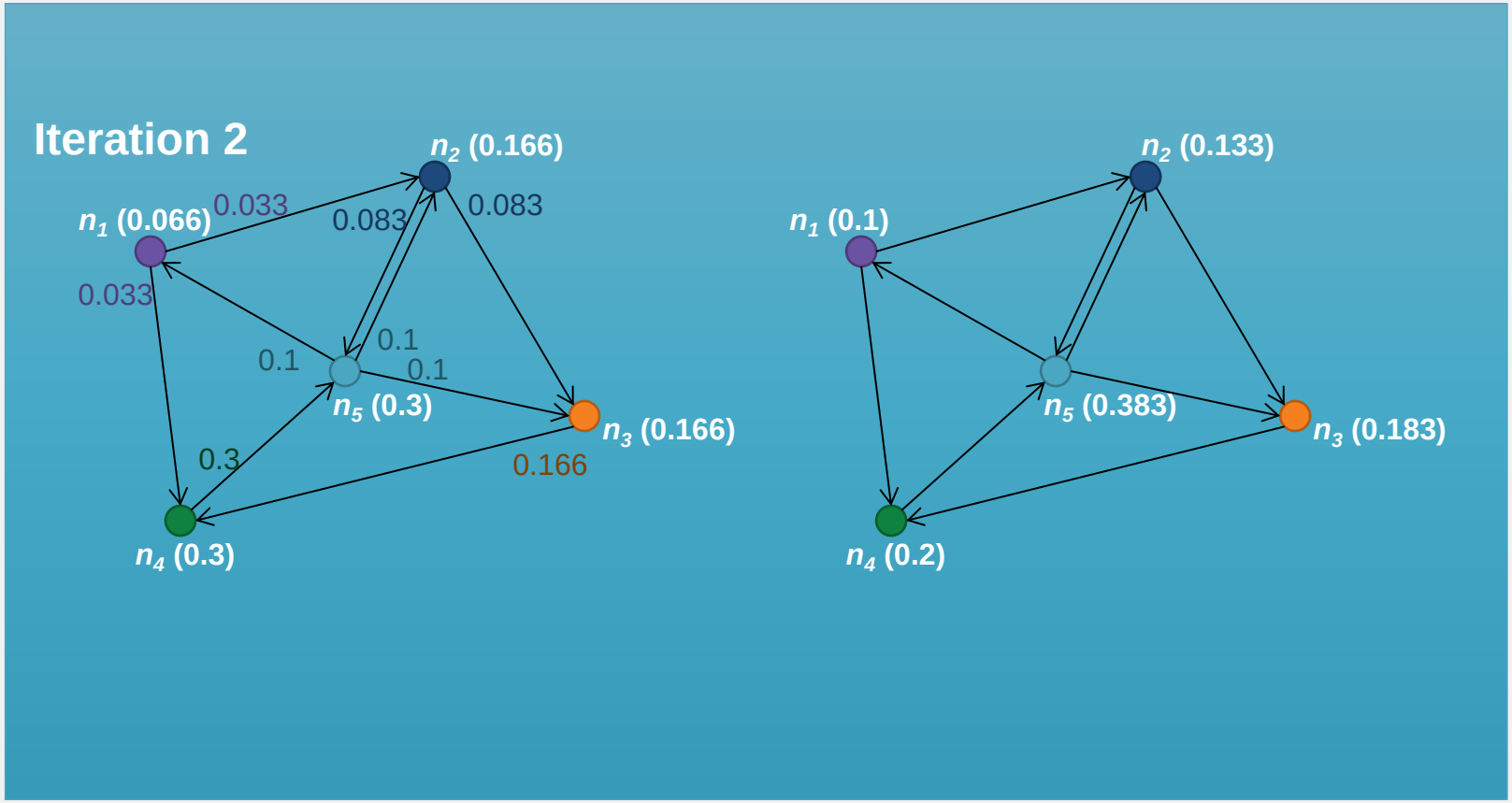
Упрощения для PageRank

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

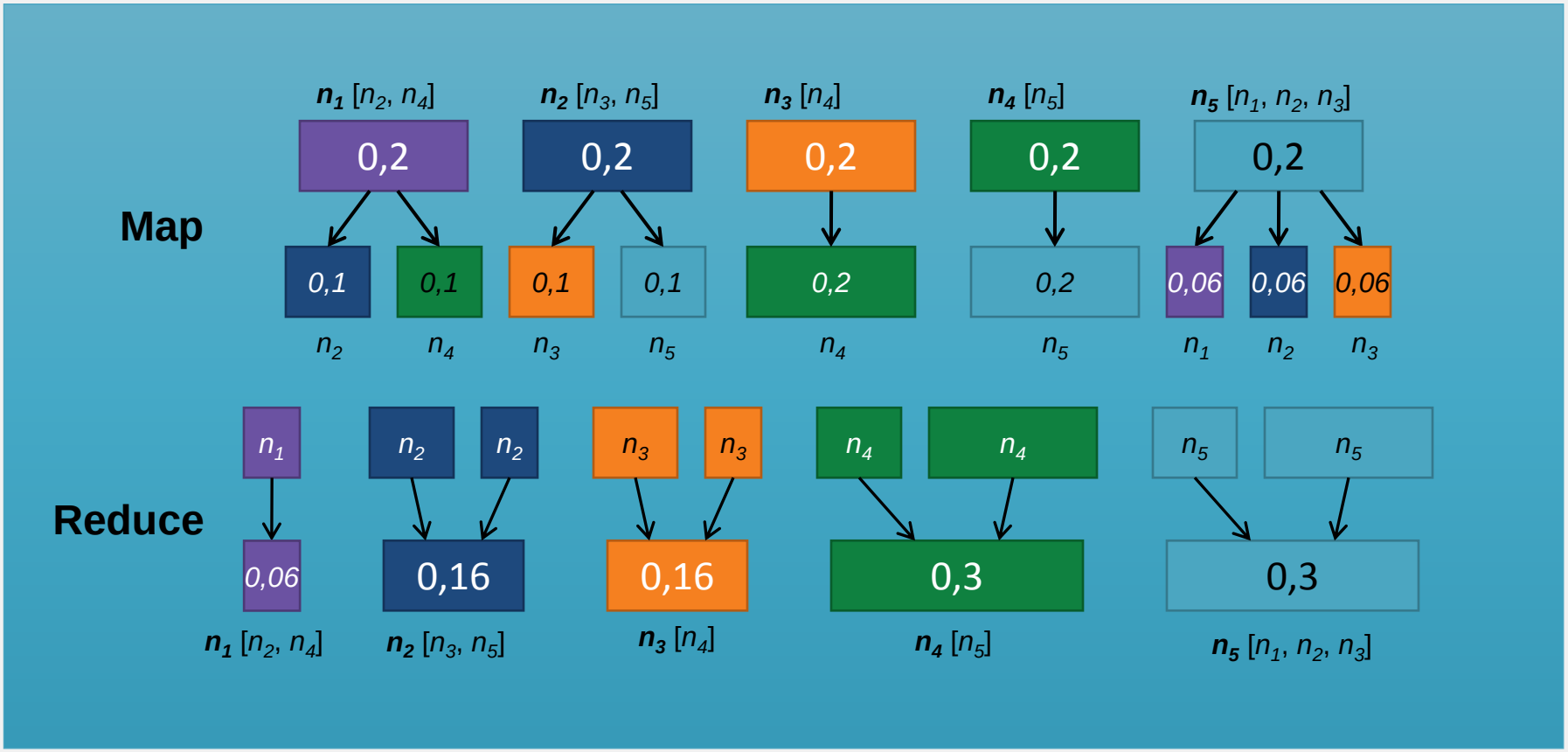
Пример расчета PageRank (1)



Пример расчета PageRank (2)



PageRank на MapReduce



PageRank: псевдокод mapper

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

Mapper 1

n_1 -> [0.2, {n_2, n_4}]

n_1 -> [0.2, {n_2, n_4}]
n_2 -> [0.1, {}]
n_4 -> [0.1, {}]

Mapper 5

n_5 -> [0.2, {n_1, n_2, n_3}]

n_5 -> [0.2, {n_1, n_2, n_3}]
n_1 -> [0.06, {}]
n_2 -> [0.06, {}]
n_3 -> [0.06, {}]

PageRank: псевдокод reducer

```
1.  class Reducer
2.      method Reduce(nid m, [p1, p2, . . .])
3.          M ← Ø
4.          for all p ∈ counts [p1, p2, . . .] do
5.              if IsNode(p) then
6.                  M ← p  // Recover graph structure
7.              else
8.                  s ← s + p // Sum incoming PageRank contributions
9.          M.PageRank ← s
10.         Emit(nid m, node M)
```

n_2 -> {[0.06, {}], [0.1, {}], [0.2, {n_3, n_5}]}

n_2 -> [0.16, {n_3, n_5}]

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

Аккуратней со ссылочным спамом:

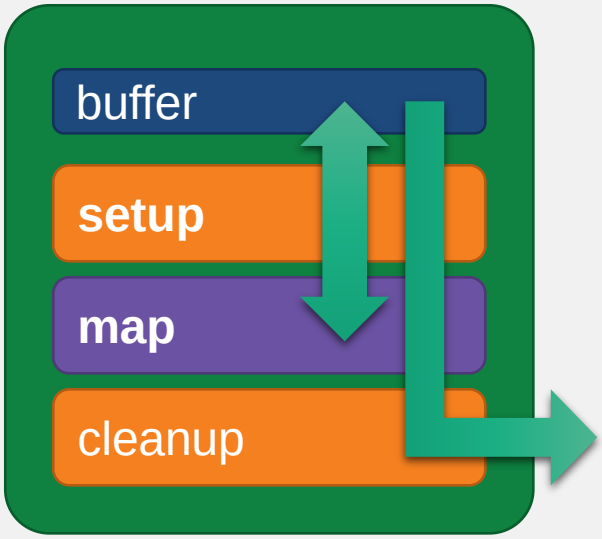
- Ссылочные фермы
- Ловушки для краулеров (*Spider traps)*
- …

MapReduce для графов – ложка дегтя

- Время запуска таска в Hadoop
- Медленные или зависшие таски
- Много обращений к диску
- 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



Emit all key-value pairs at once

Улучшение партиционирования

По-умолчанию: hash partitioning

- Произвольно присвоить вершину к партиции

Наблюдение: много графов имеют локальную структуру

- Например, 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

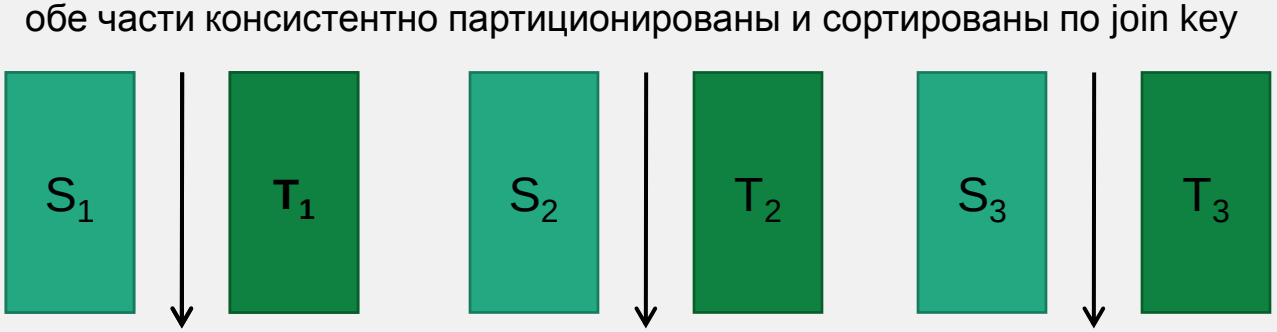
Schimmy Design Pattern

Основная реализация содержит два набора данных:

- *Messages* (актуальные вычисления)
- *Graph structure* (структура обрабатываемого графа)

Schimmy: разделить два набора данных, выполнять shuffle только для *messages*

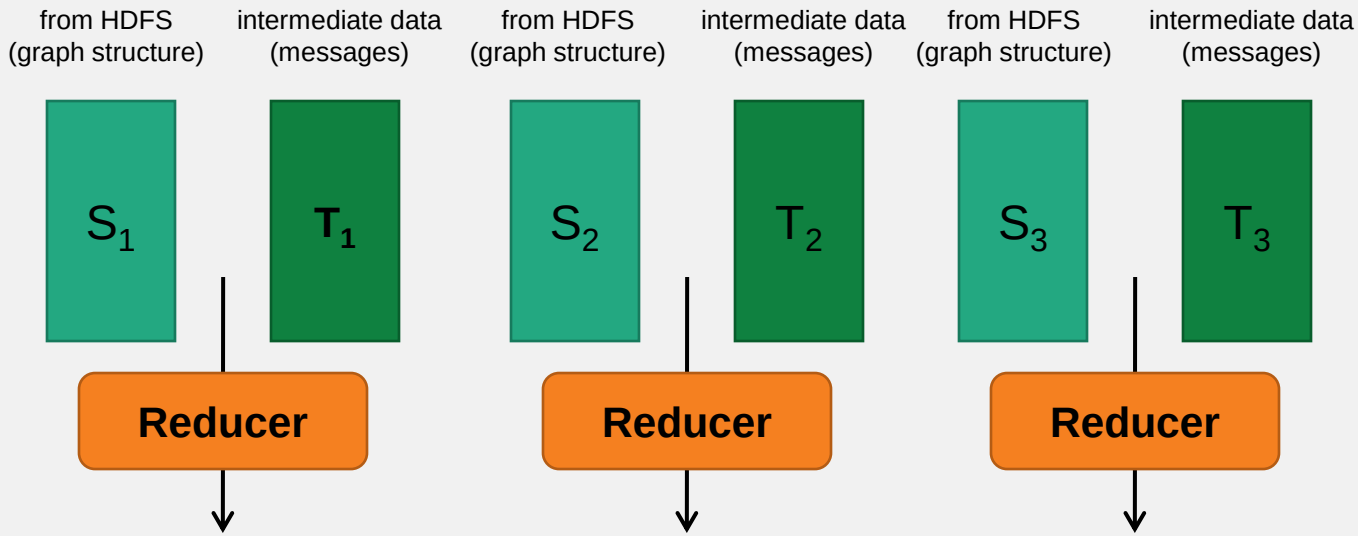
- Основная идея: выполнять *merge join* для *graph structure* и *messages*



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

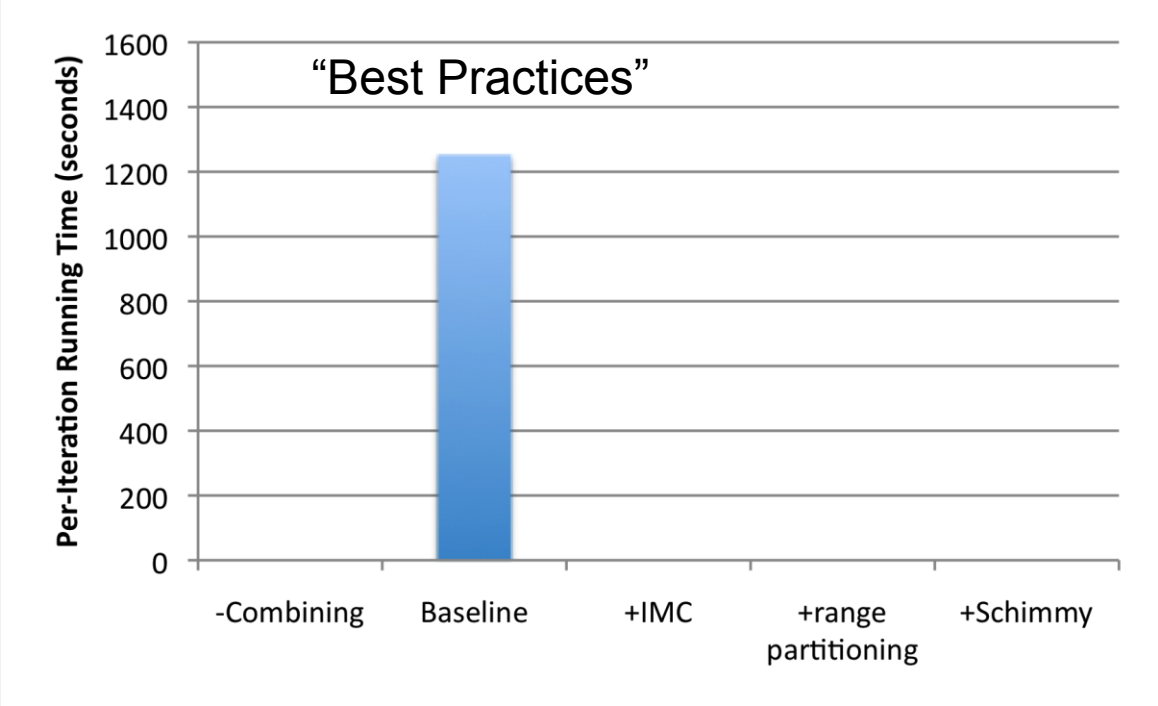
Dataset:

- Первый сегмент английского текста из 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

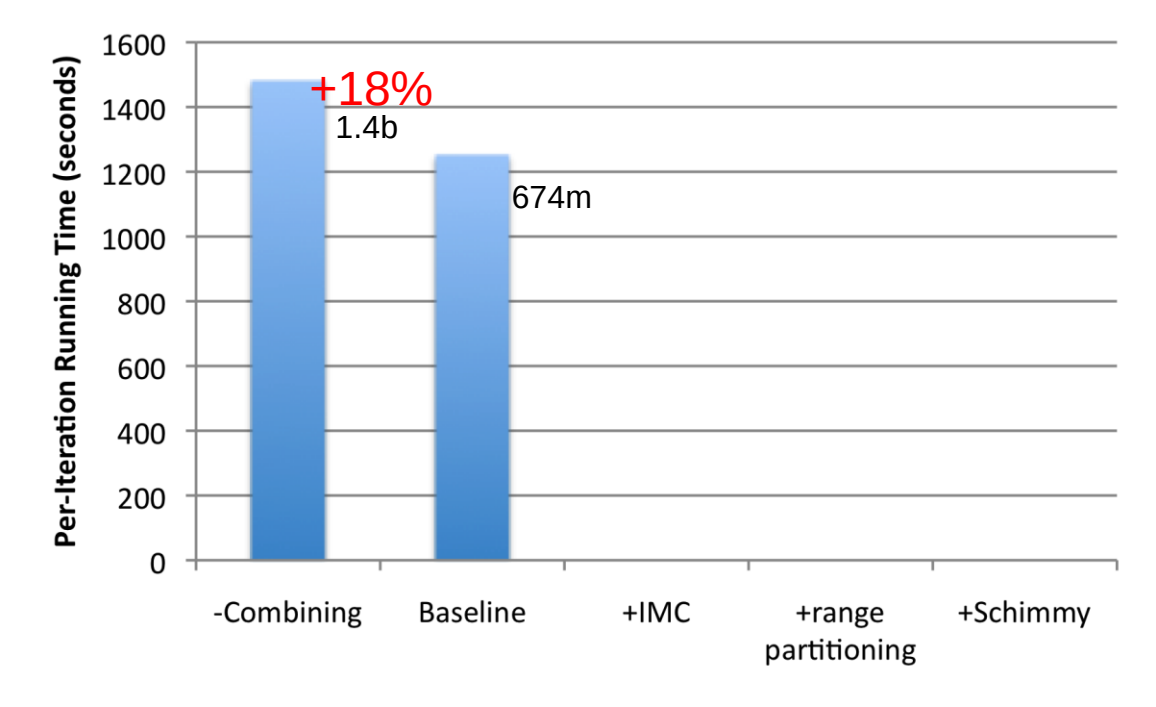
Setup:

- Измерялось время выполнения по каждой итерации (5 итераций)
- 100 партиций

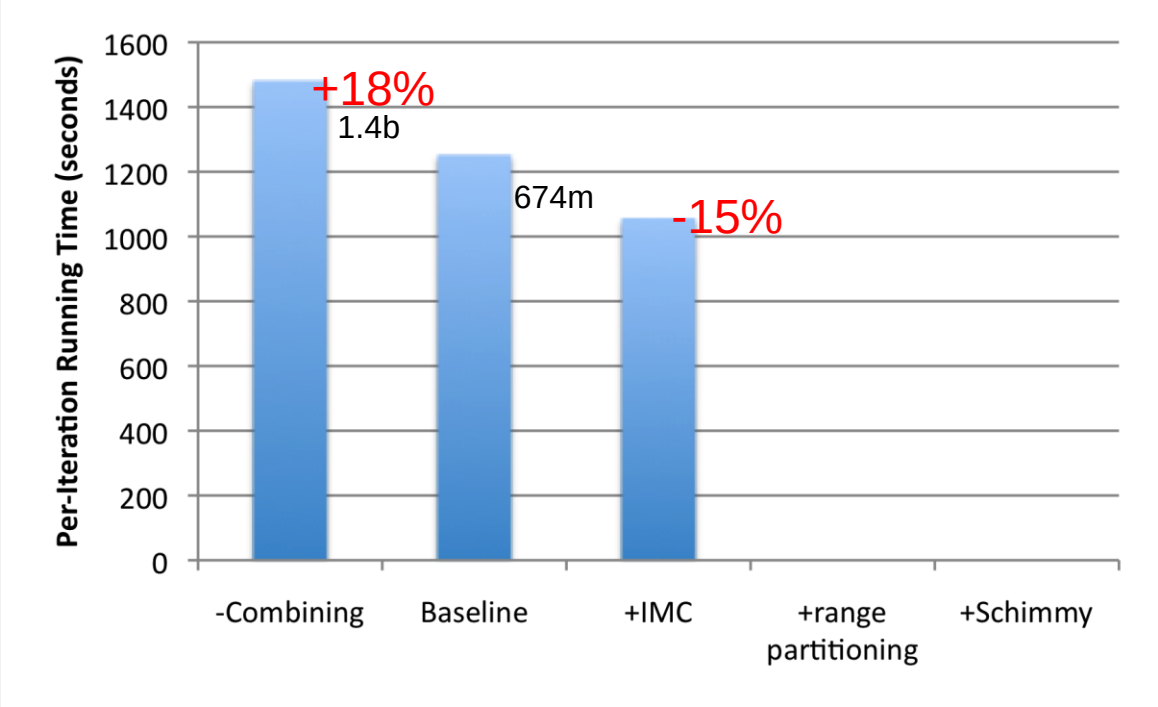
Результаты


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

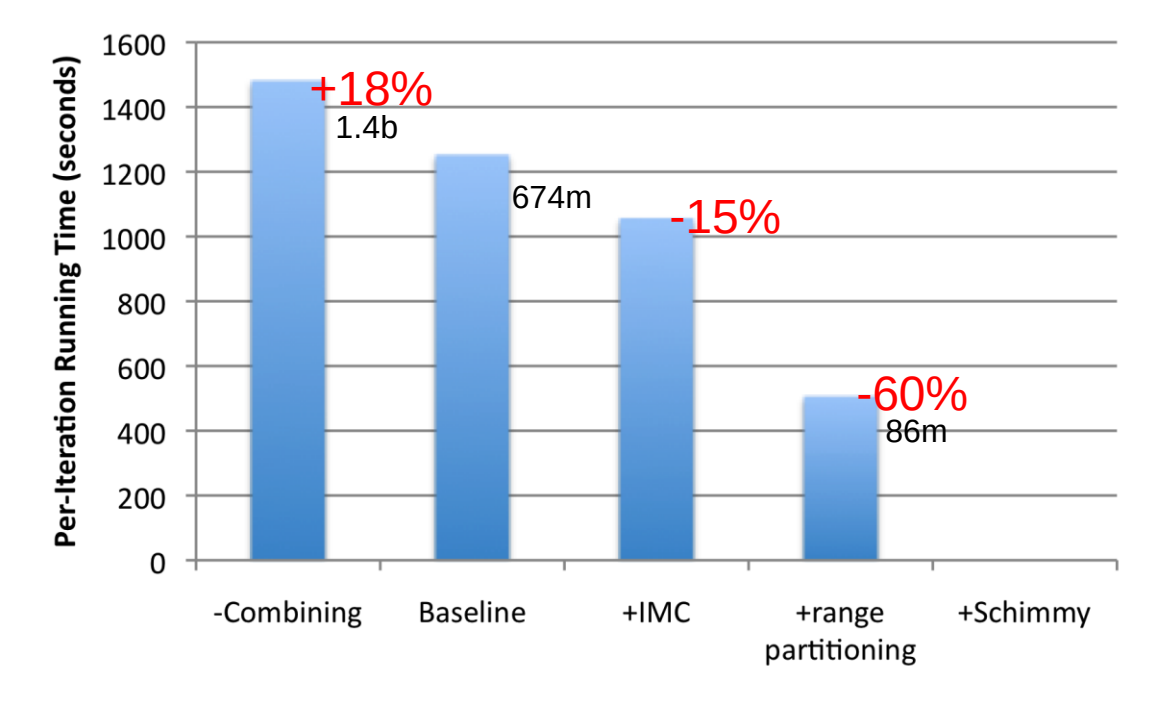
Результаты


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

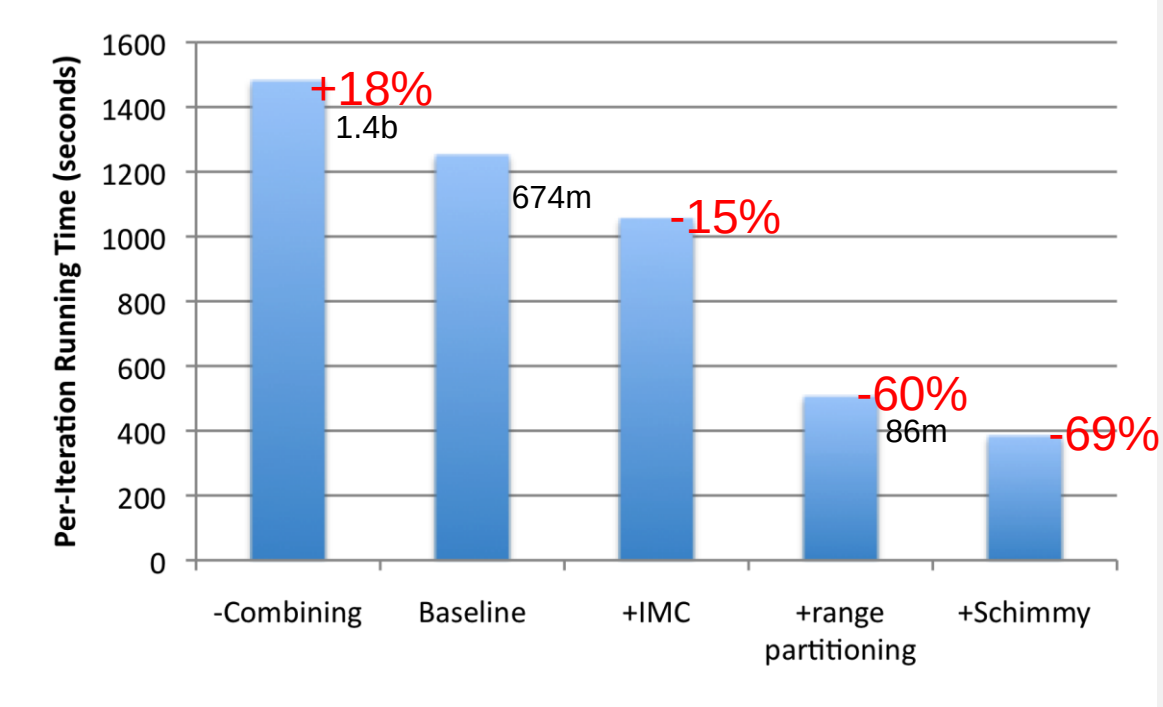
Результаты


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


APACHE
GIRAPH

Apache Giraph: история

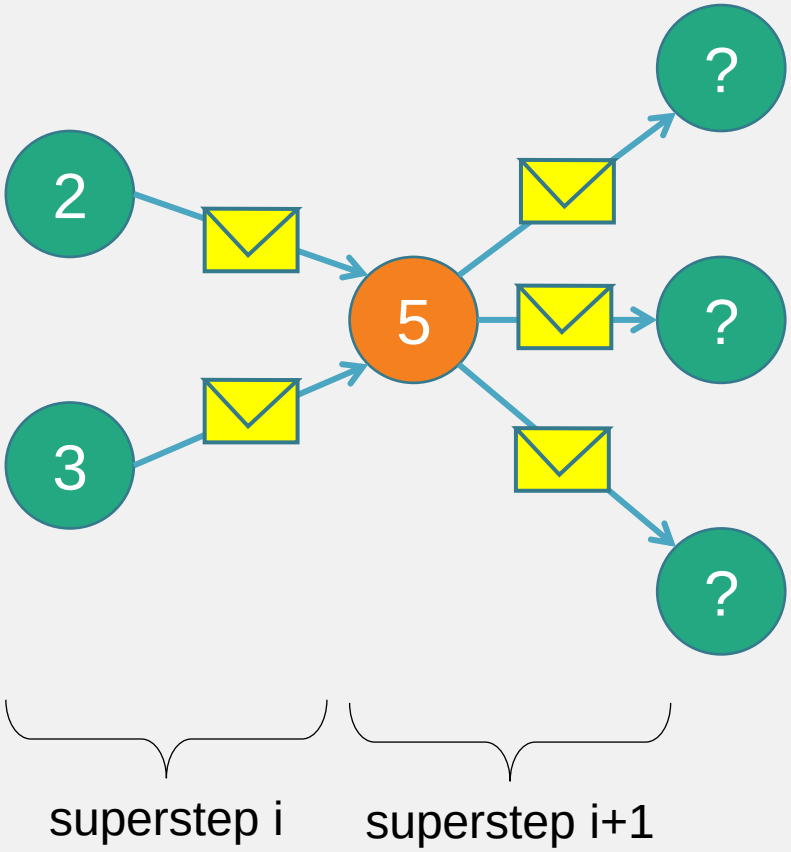
- Google Pregel (2010)
  - https://kowshik.github.io/JPregel/pregel_paper.pdf
- Yahoo! инвестировал в разработку (2011)
- Проект верхнего уровня (2012): giraph.apache.org
- Релиз 1.0 (май 2013)
- Релиз 1.1 (ноябрь 2014)

Apache Giraph: основные принципы

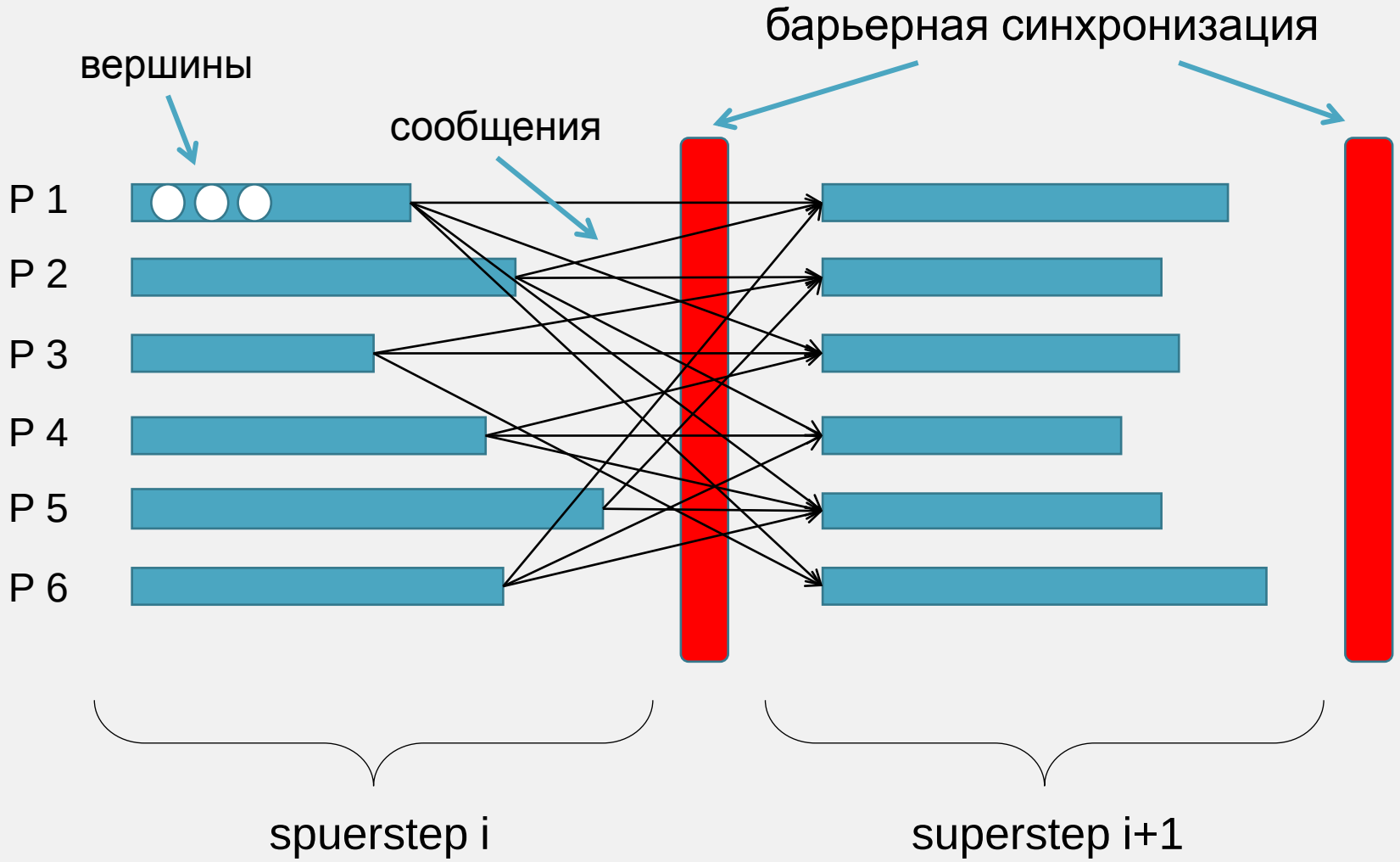
- Распределенные вычисления на большом графе
- “Думай как вершина” (**vertex-centric API**)
- Вычисления состоят из итераций (**superstep**)
- Синхронизация происходит по паттерну **Bulk Synchronous Parallel** (BSP)
- Отказоустойчивость (**checkpoint**)

Apache Giraph: Vertex-centric

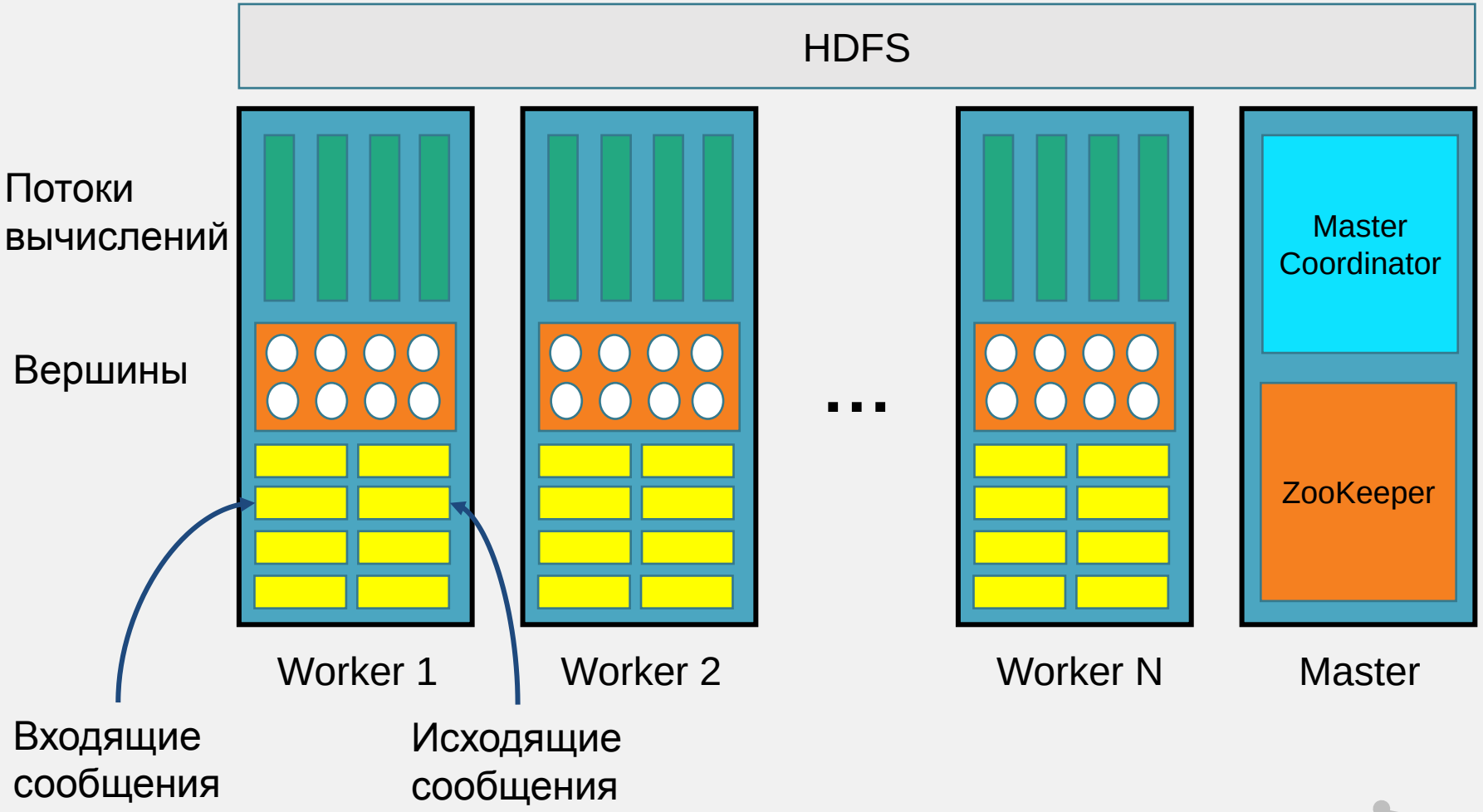
- Вершина имеет значение и список исходящих ребер
- 2 состояния: **active** и **vote to halt**
- Общение между вершинами через сообщения (**message**)
- Вычисления реализуются в функции **compute**(). Она выполняется для каждой вершины на каждой итерации



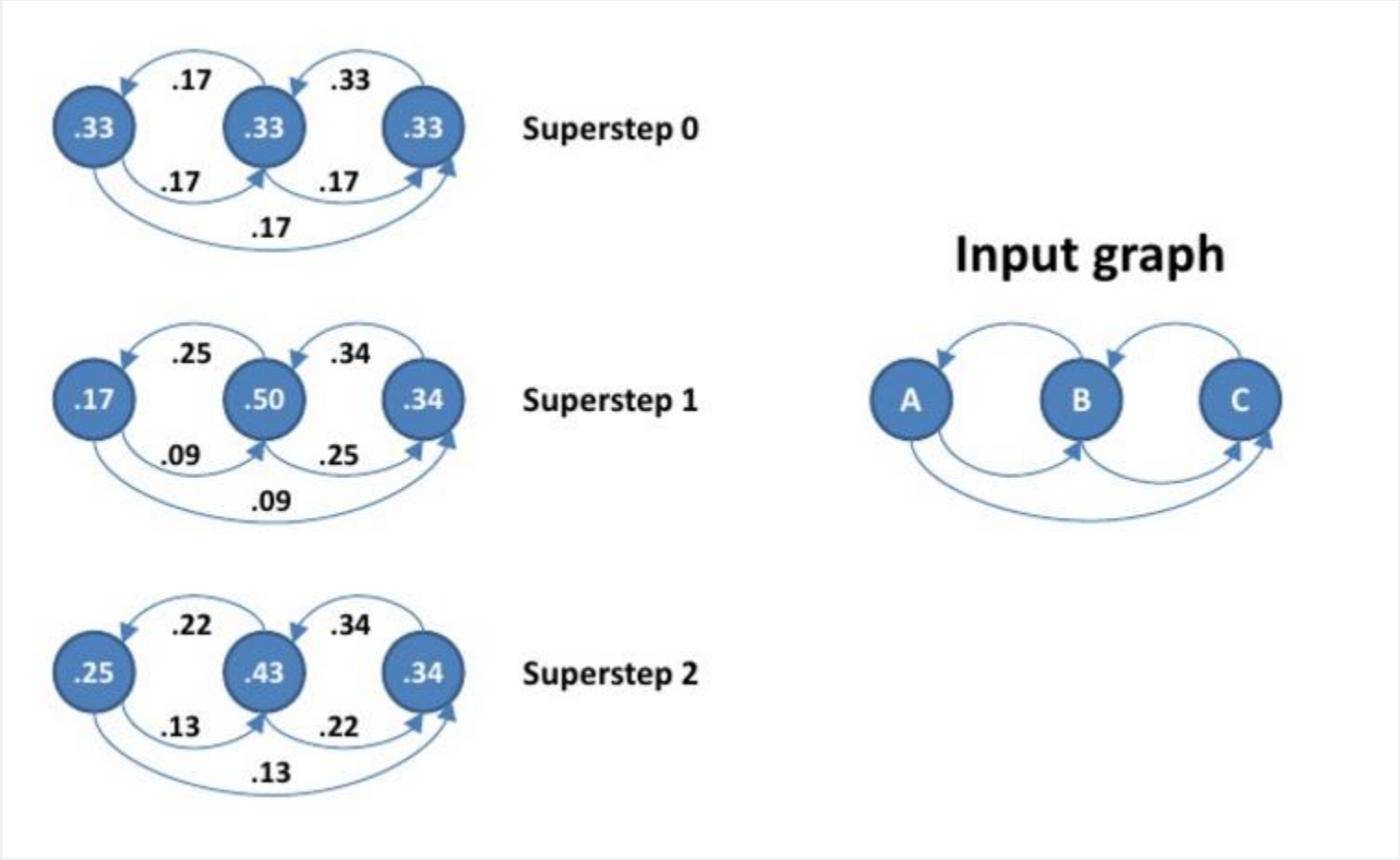
Apache Giraph: BSP



Apache Giraph: Архитектура



Apache Giraph: PageRank



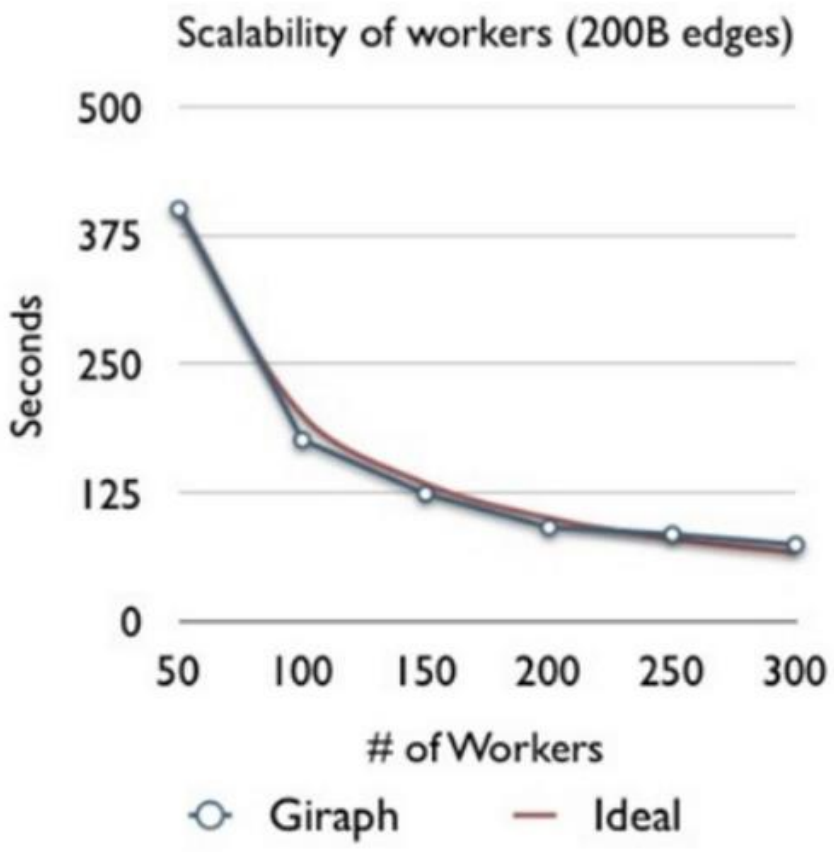
Apache Giraph: PageRank

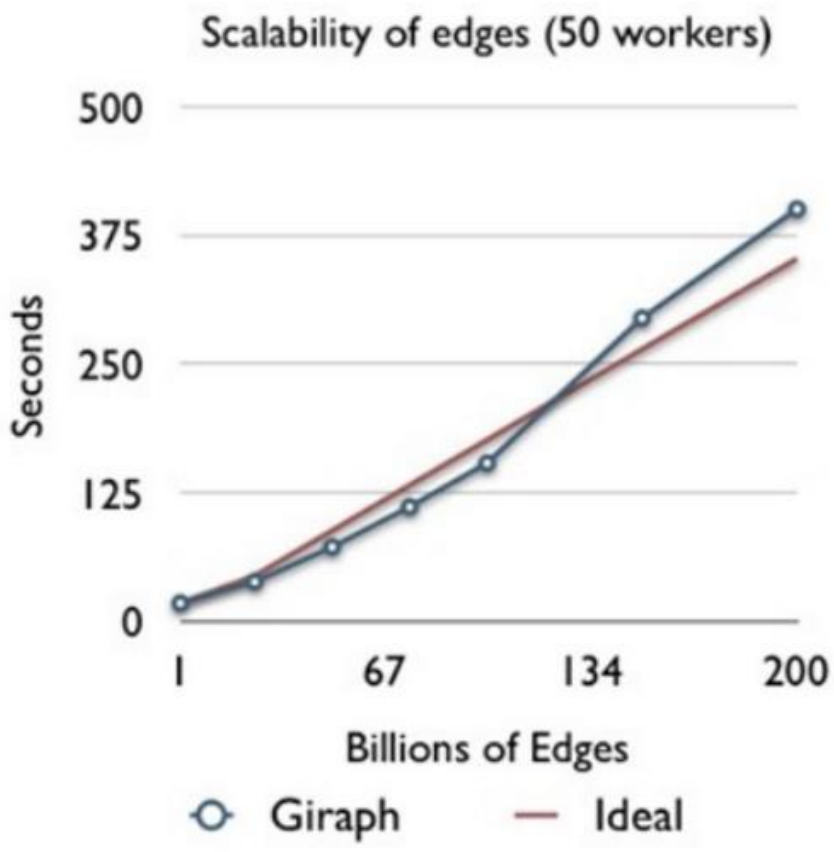
```
class PageRankVertex {
  void compute(Iterator messages) {
    if (getSuperstep() > 0) {
      // recompute own PageRank from the neighbors messages
      pageRank = sum(messages);
      setVertexValue(pageRank);
    }

    if (getSuperstep() < k) {
       // send updated PageRank to each neighbor
       sendMessageToAllNeighbors(pageRank / getNumOutEdges());
    } else {
      voteToHalt();  // terminate
    }
}}
```

$$p_i = \sum_{j \in \{(j,i)\}} \frac{p_j}{d_j}$$

Apache Giraph: Масштабируемость



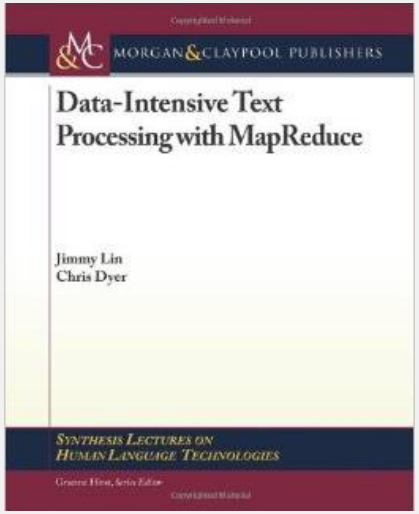


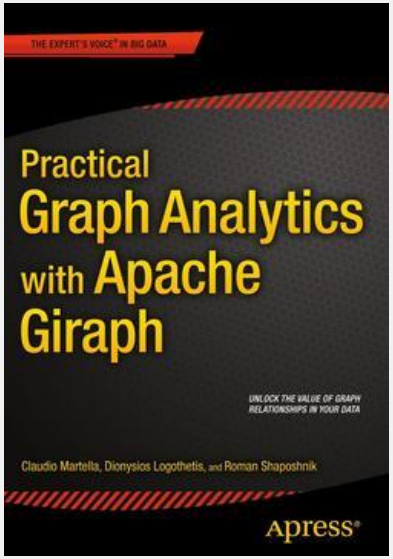
Apache Giraph: дополнительные возможности

- **Combiner**:
  - Проблема: отправка сообщений – дорогая операция
  - Решение: скомбинировать их перед отправкой
- **Aggregator**:
  - Вершины отправляют значение на итерации S
  - Giraph комбинирует эти значения
  - Полученное значение доступно вершинам на S+1 итерации

Ресурсы

- **Data-Intensive Text Processing with MapReduce**
- Jimmy Lin and Chris Dyer (Authors) (April, 2010)
- Chapter5: Graph Algorithms





- **Practical Graph Analytics with Apache Giraph**
- by Dionysios Logothetis, Roman Shaposhnik, Claudio Martella) (*October 2015*)

ТЕХНОСФЕРА

Отмечайтесь и оставляйте отзыв

Спасибо за внимание!

Евгений Чернов

e.chernov@corp.mail.ru