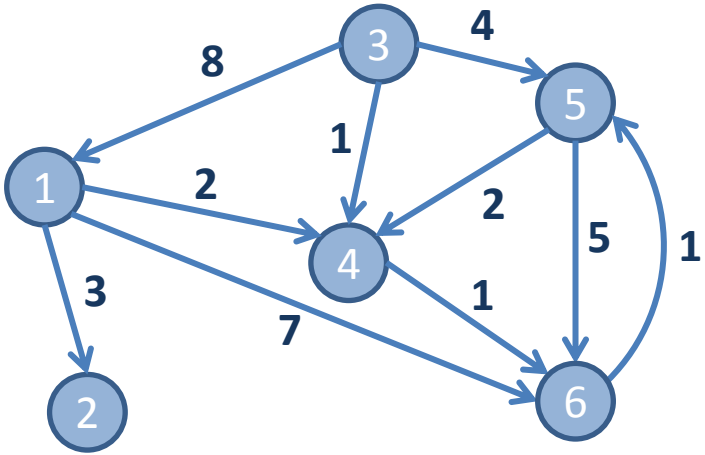
MapReduce в Hadoop
Графы

Граф как структура данных

G = (V,E)

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

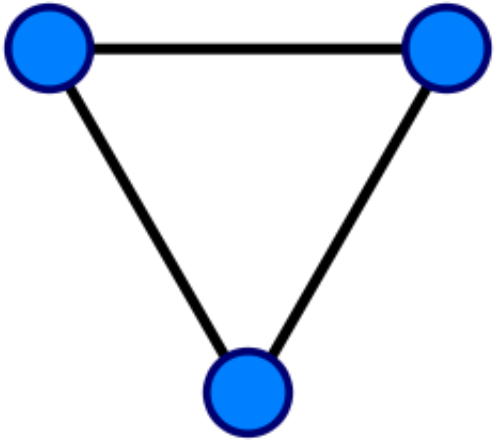


$V = \{1, 2, 3, 4, 5, 6\}$

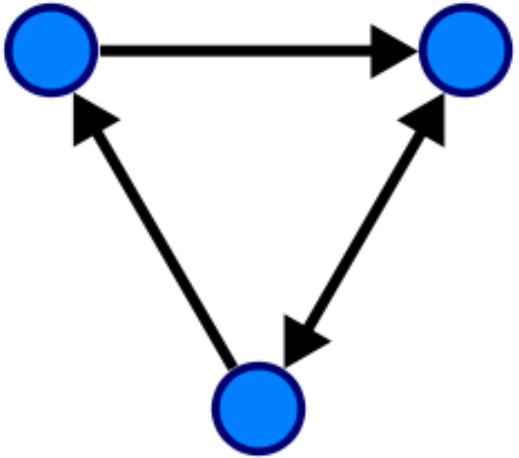
$E = \{[1,2], [1,4], [1,6], [3,1], \ldots\}$

$W_{1,2} = 3,\ W_{1,4} = 2,\ W_{1,6} = 7,\ W_{3,1} = 8,\ \ldots$

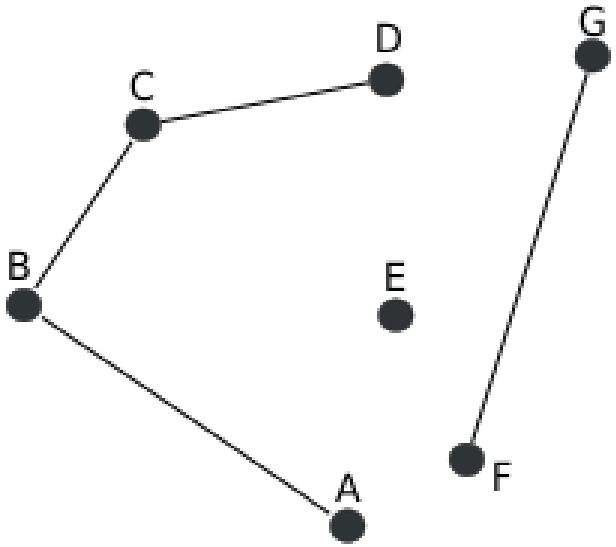
Виды графов



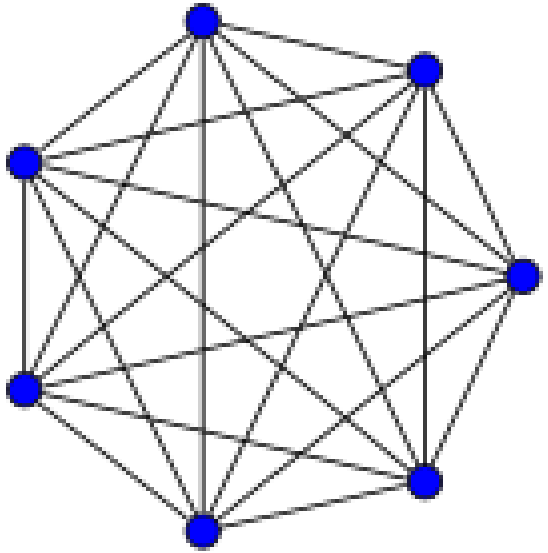
Неориентированный



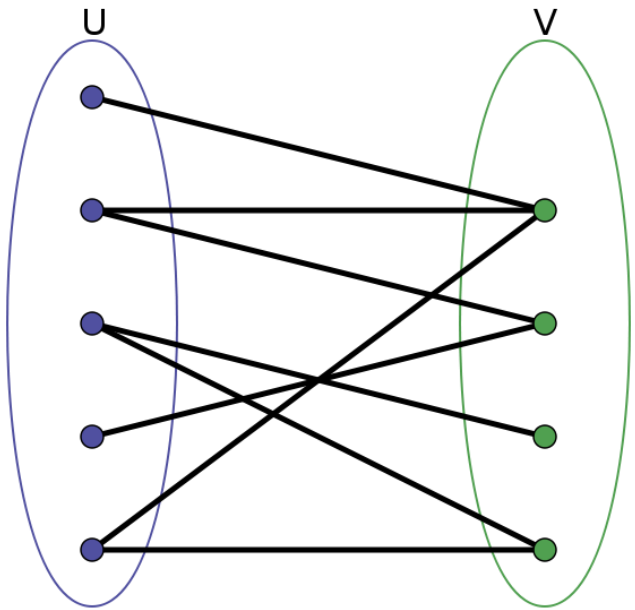
Ориентированный



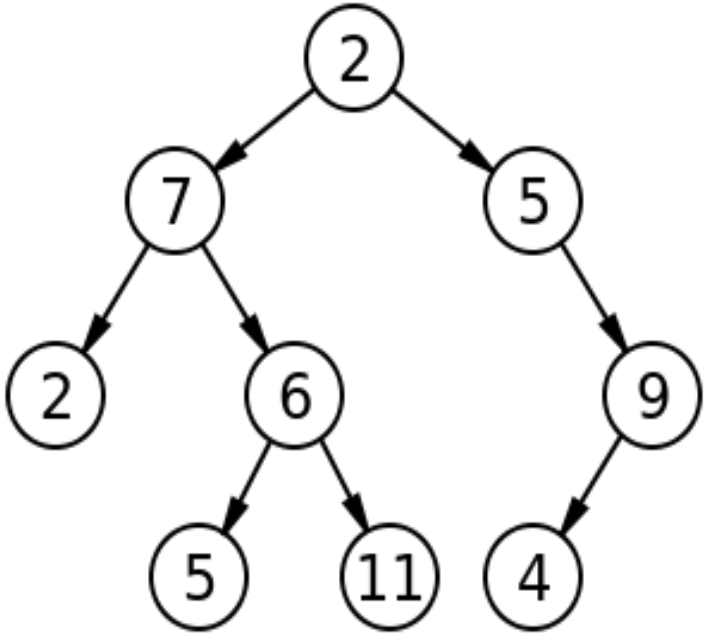
Несвязный



Полный

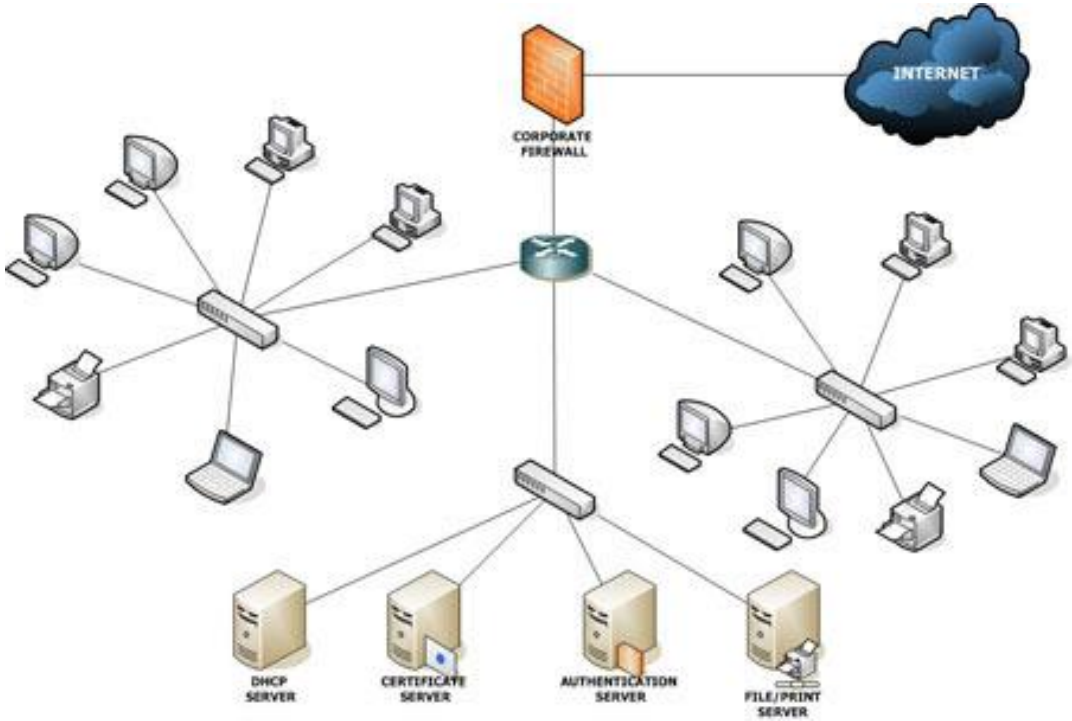


Двудольный

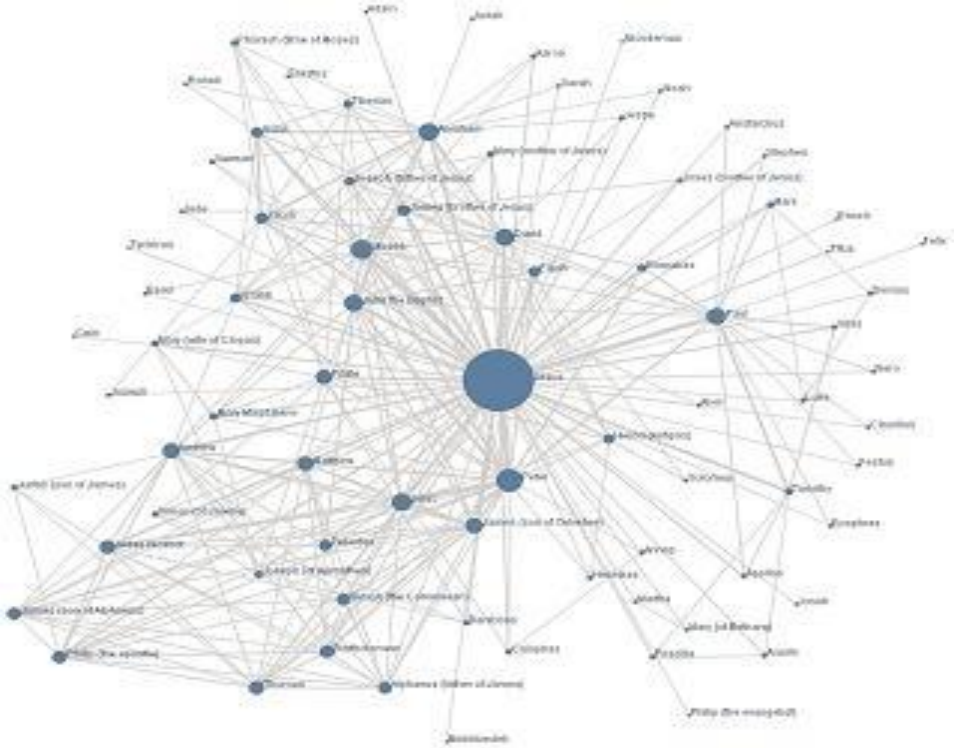


Дерево

Графы на практике



Структура компьютеров и серверов сети



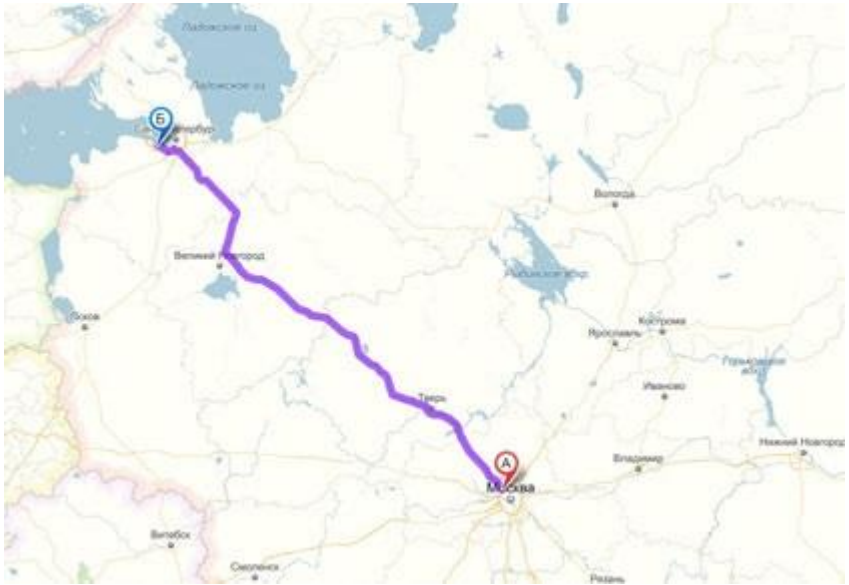
Социальные сети


9
Алтуфьево
Бибирево
Отрадное
Владыкино
Петровско-Разумовская
Тимирязевская
Дмитровская
Савеловская
Минделеевская
Новослободская
Цветной бульвар
Чеховская
Боровицкая
Полянка
Серпуховская
Тульская
Нагатинская
Нагорная
Нахимовский пр.
Севастопольская
Чертановская
Южная
Пражская
Улица Академика Янгеля
Аннино
Бульвар Дмитрия Донского
Старокачаловская
Скобелевская
Адмирала Ушакова
Бунинская Аллея
Гончарная
2
Речной канал
Водный тсдион
Войковская
Сокол
Аэропорт
Динамо
Белорусская
Маяковская
Тверская
Театральная
Новокузнецкая
Павелецкая
Автозаводская
Коломенская
Каширская
Кантемировская
Царицыно
Орехово
Домодедовская
Красногвардейская
3
Митино
Волоколамская
Мякинино
Строгино
Крылатское
Молодежная
Кунцевская
Славянский бульвар
Парк победы
Киевская
Смоленская
Арбатская
Площадь Революции
Курская
Бауманская
Электрозаводская
Семеновская
Партизанская
Измайловская
Первомайская
Щелковская
Пионерская
Филевский парк
Багратионовская
Фили
Кутузовская
Студенческая
Международная
Выставочная
Александровский сад
7
Планерная
Сходненская
Тушинская
Щукинская
октябрьское п.
Полежаевская
Беговая
ул. 1905 года
Баррикадная
Пушкинская
Кузнецкий мост
Китай - город
Таганская
Пролетарская
Волгоградский пр.
Текстильщики
Кузьминки
Рязанский пр.
Выхино
1
Улица Подбельского
Черкизовская
Преображенская площадь
Сокольники
Красносельская
Комсомольская
Красные Ворота
Чистые пруды
Лубянка
Охотный ряд
Библиотека им. Ленина
Кропоткинская
Парк культуры
Фрунзенская
Спортивная
Воробьевы горы
Университет
Пр. Вернандского
Юго-Западная
Краснопресненская
Октябрьская
Добрынинская
6
Медведково
Бабушкинская
Свиблово
Ботанический сад
ВДНХ
Алексеевская
Рижская
пр. Мира
Сухаревская
Тургеневская
Третьяковская
Шаболовская
Ленинский пр.
Академическая
Пфсоюзная
Новые Черемушки
Калужская
Беляево
Коньково
Теплый Стан
Ясенево
Новоясеневская
8
Марьина роща
Достоевская
Трубная
Сретенский бульвар
Чкаловская
Римская
Крестьянская застава
Дубровка
Кожуховская
Печатники
Волжская
Люблино
Братиславская
Марьино
Борисово
Шипиловская
Зябликово
Новогиреево
Перово
ш. Энтузиастов
Авиамодельная
Площадь Ильича
Макрксистская
Варшавская
Каховская

Схема метро

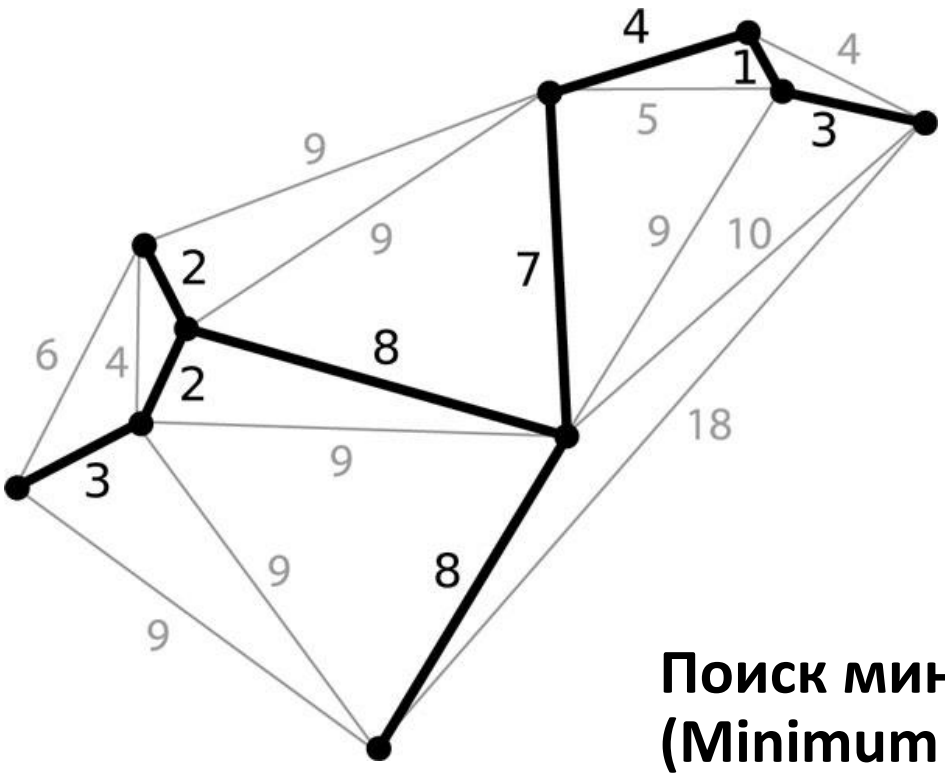
Структура дорог

Задачи и проблемы на графах



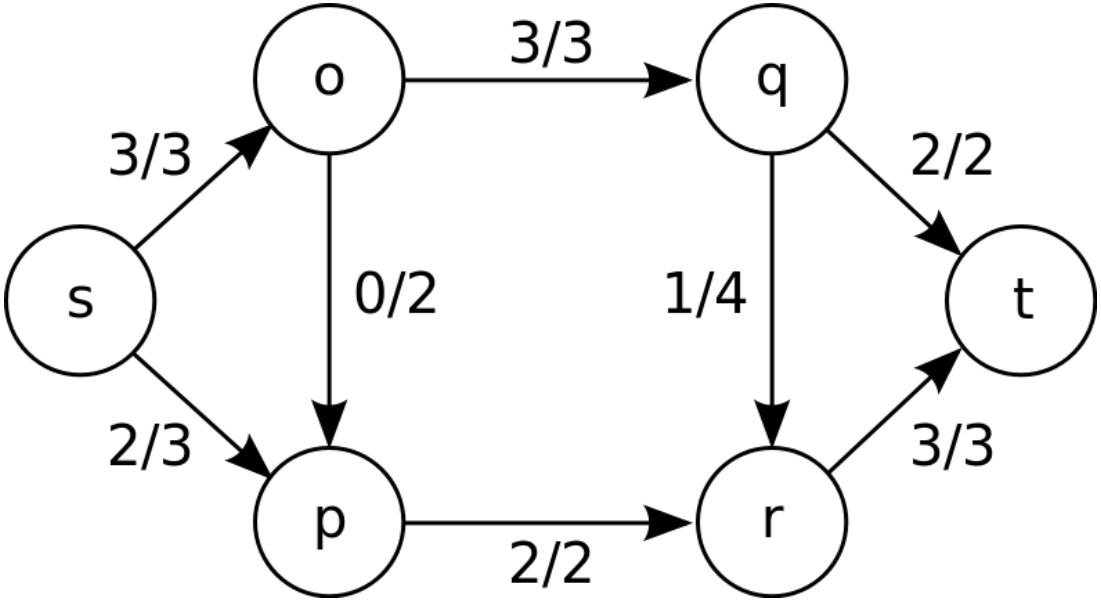
Поиск кратчайшего пути

- Роутинг траффика
- Навигация маршрута



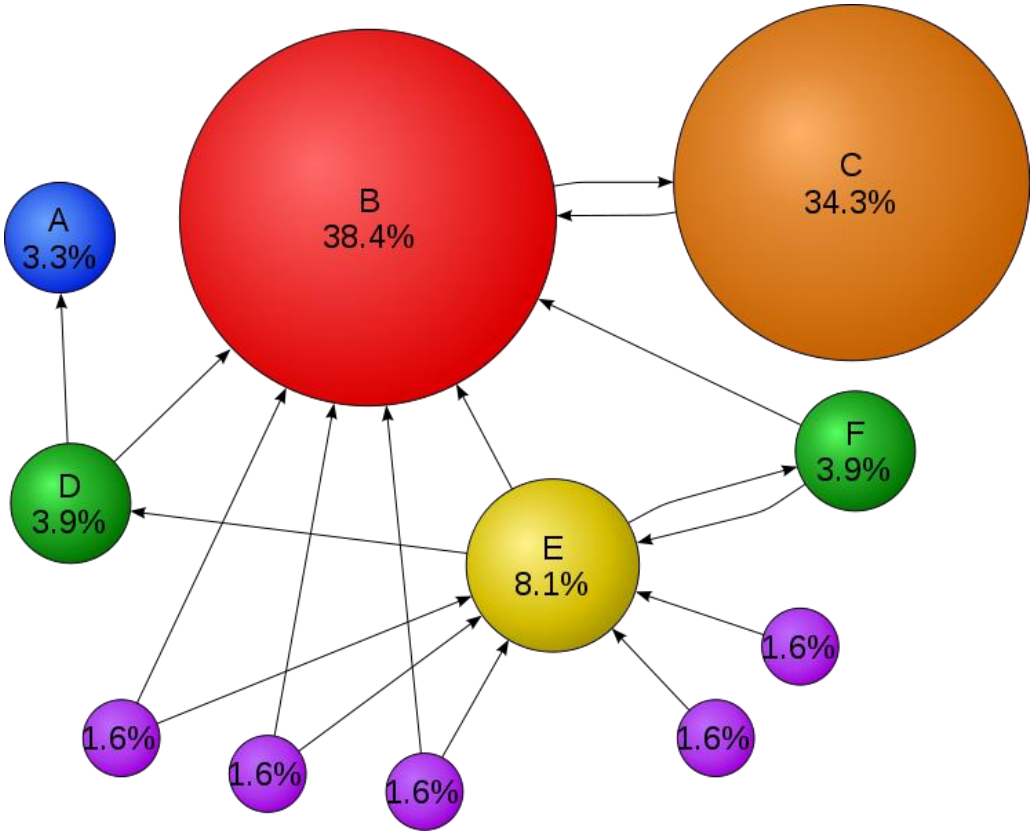
Поиск минимального остовного дерева (Minimum Spanning Tree)

- Телекоммуникационные компании



Поиск максимального потока (Max Flow)

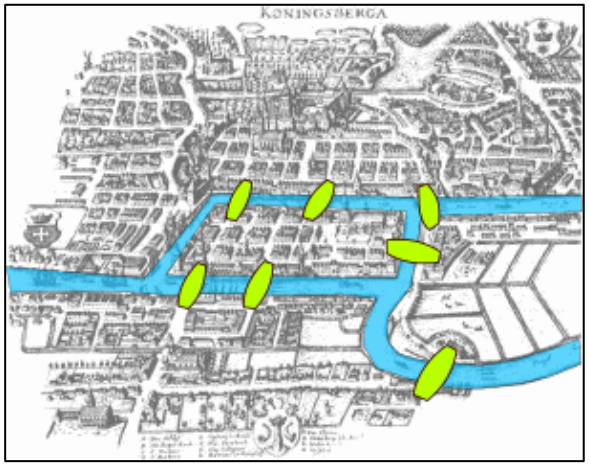
- Структура компьютеров и серверов Интернет



Алгоритмы ссылочного ранжирования

- PageRank
- HITS

Проблема семи мостов Кёнигсберга



Леонард Эйлер

Графы и MapReduce

- Вычисления на каждой вершине
- Обход графа

Ключевые вопросы:

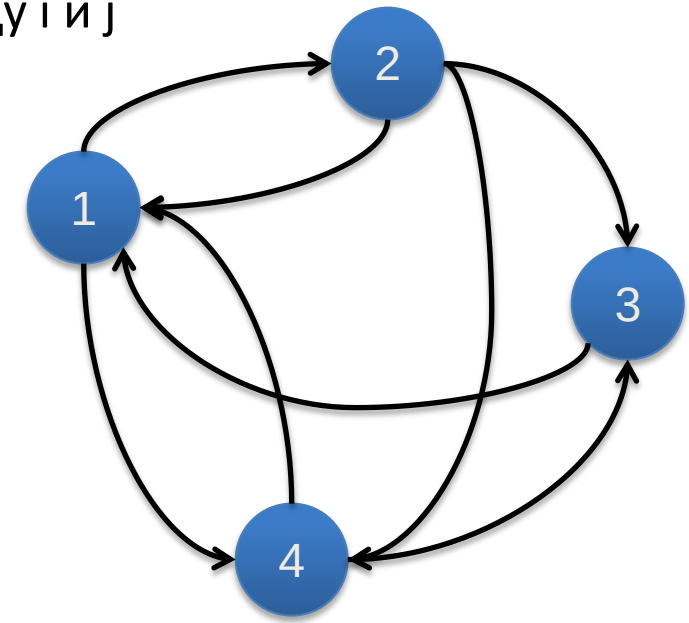
- Как представить граф в MapReduce?
- Как обходить граф в MapReduce?

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

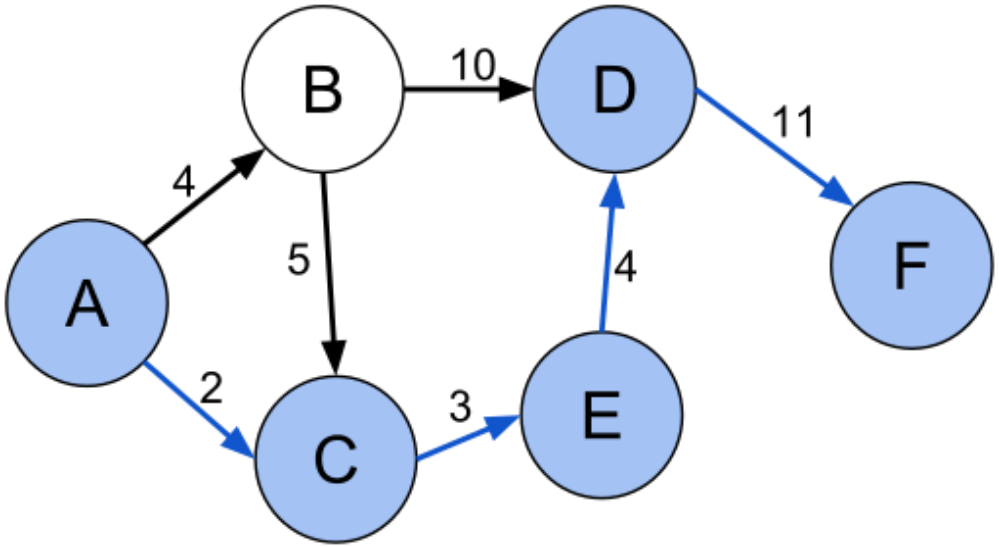
	1	**2**	**3**	**4**
1	0	1	0	1
2	1	0	1	1
3	1	0	0	0
4	1	0	1	0

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

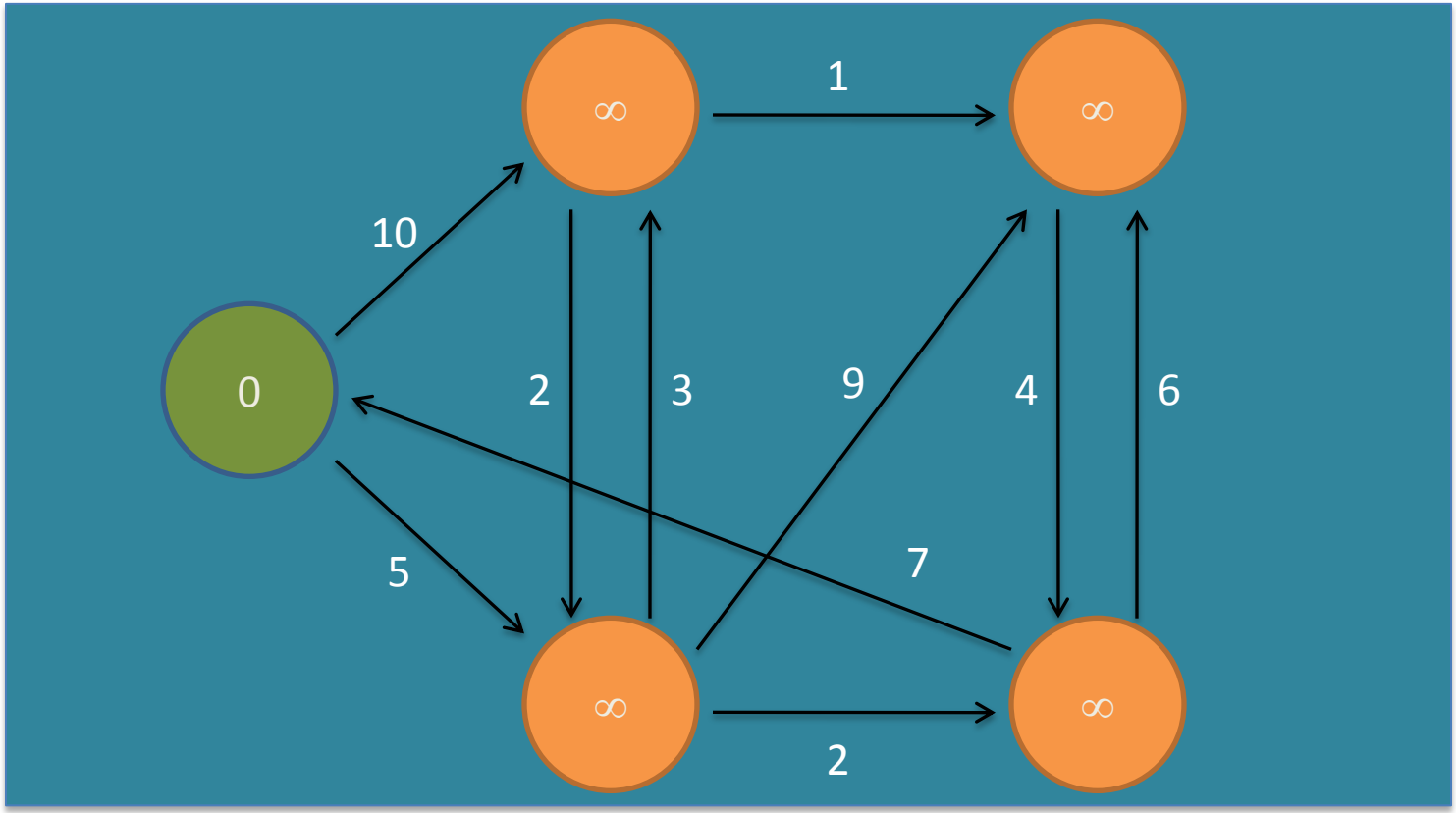
Списки смежности

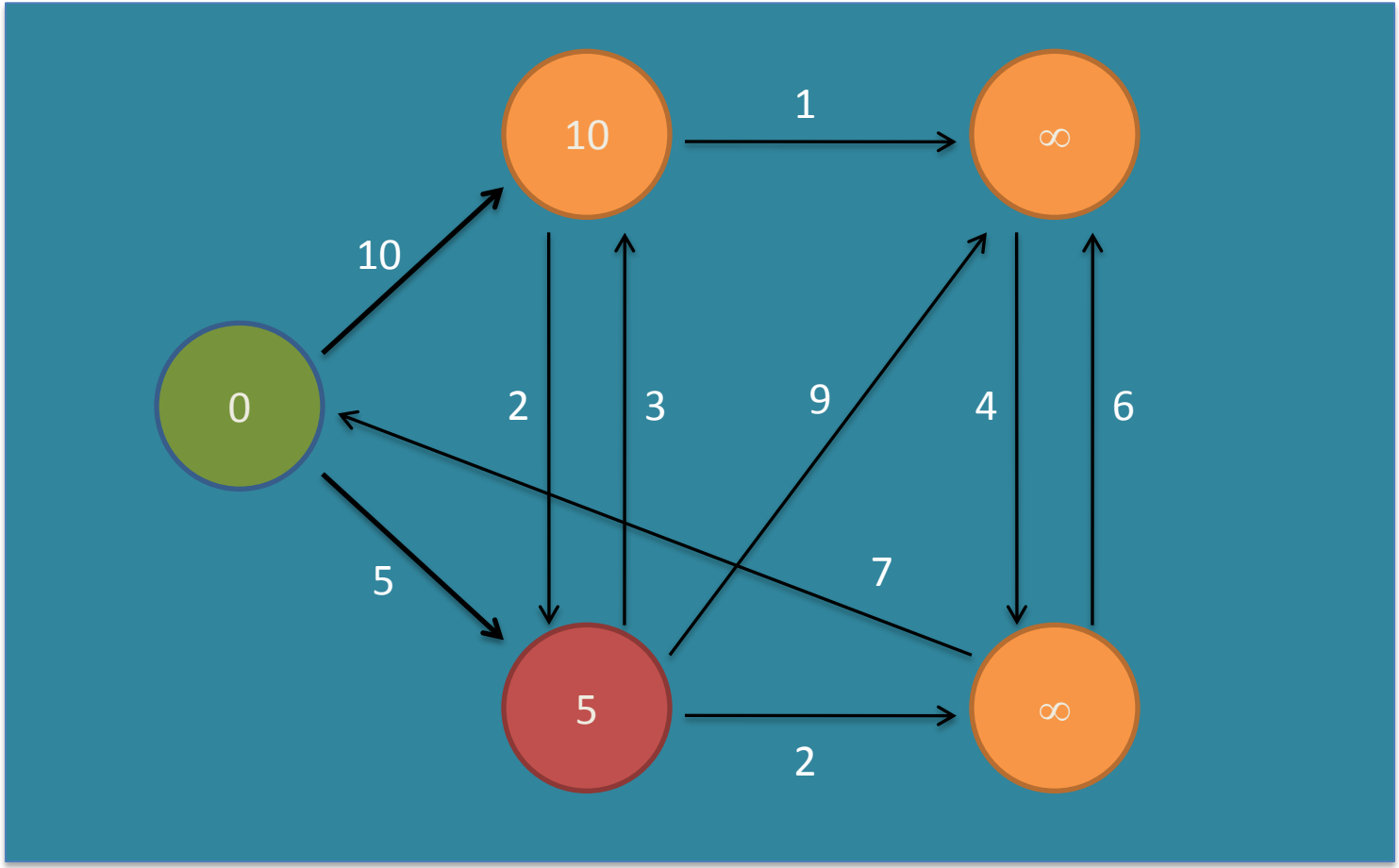
- **Плюсы**
  - Намного более компактная реализация
  - Легко найти все исходящие ссылки для вершины
- **Минусы**
  - Намного сложнее подсчитать входящие ссылки

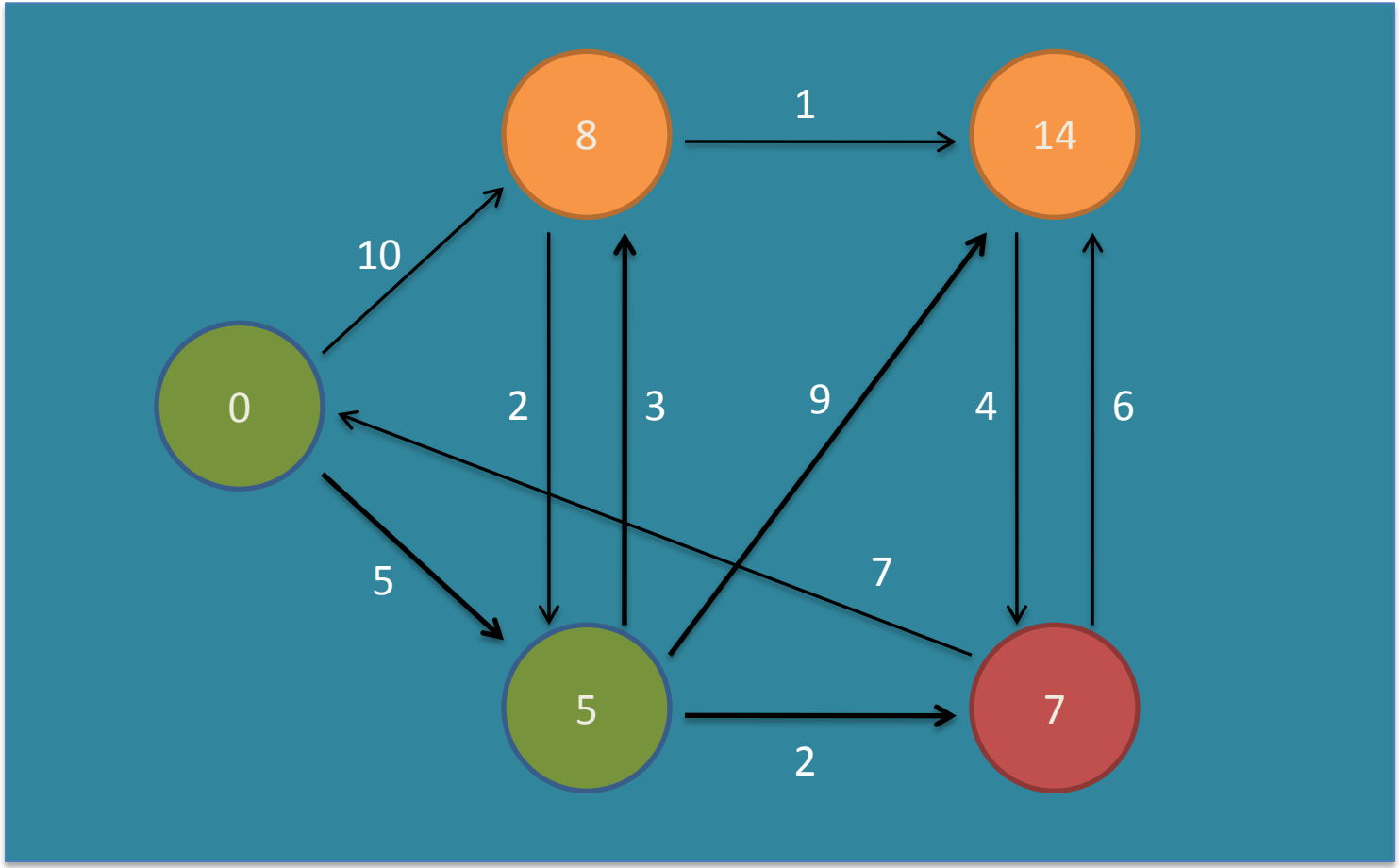
Поиск кратчайшего пути в графе

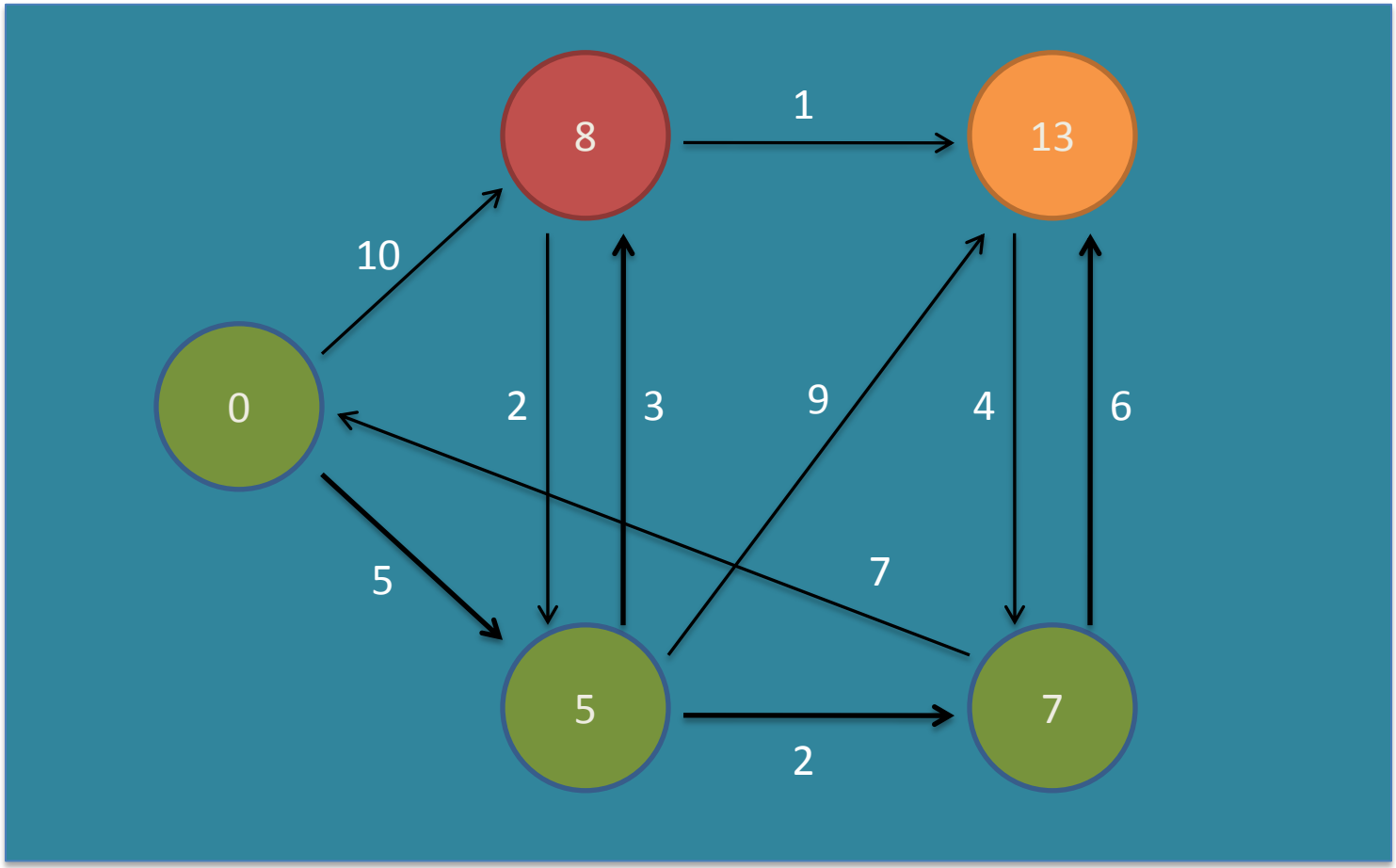


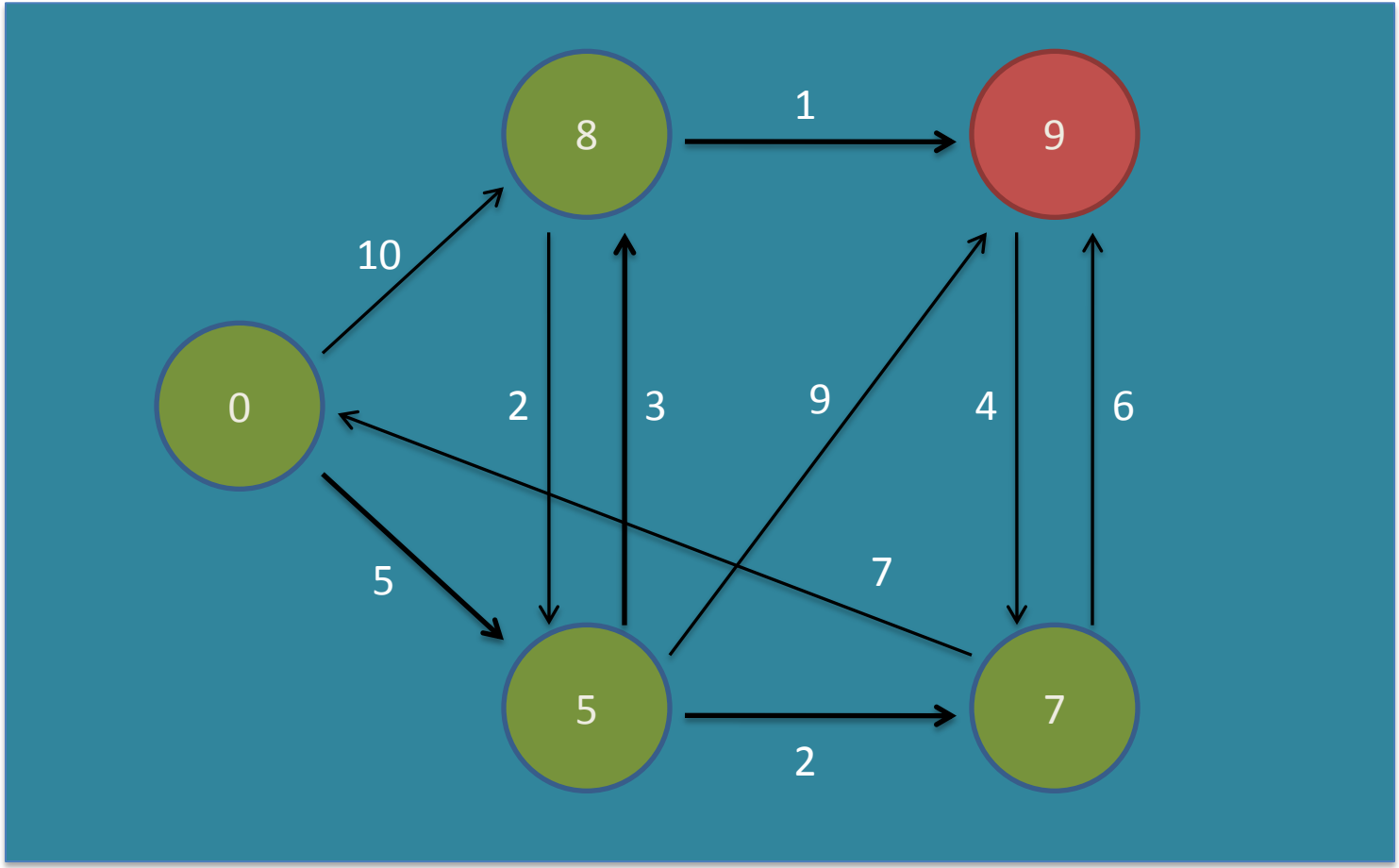
Алгоритм Дейкстры

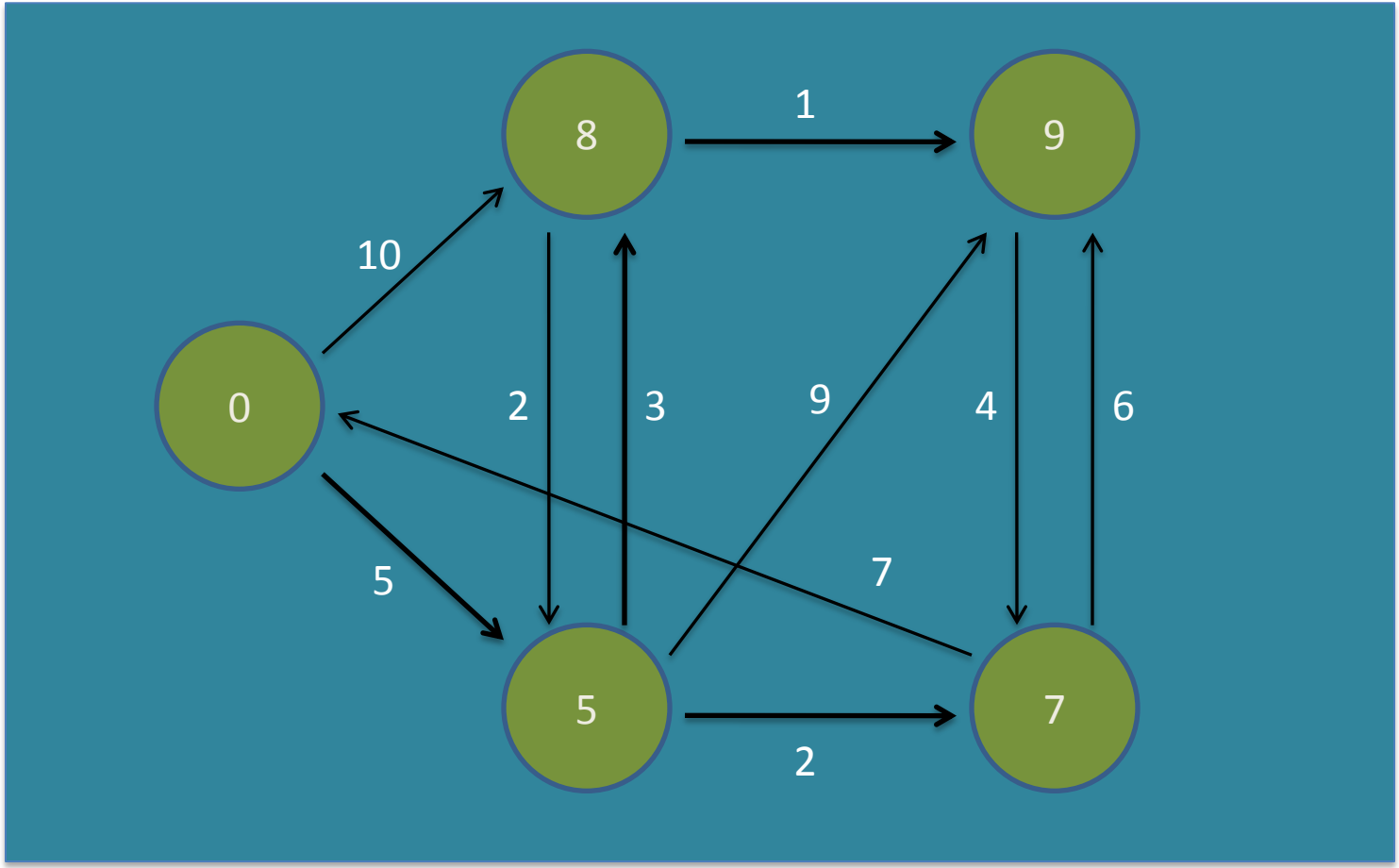



10
1
∞
10
0
2
3
9
4
6
5
7
5
∞
2


8
1
14
10
0
2
3
9
4
6
7
5
5
7
2


8
1
13
10
0
2
3
9
4
6
7
5
5
2
7


8
1
9
10
0
2
3
9
4
6
7
5
5
7
2

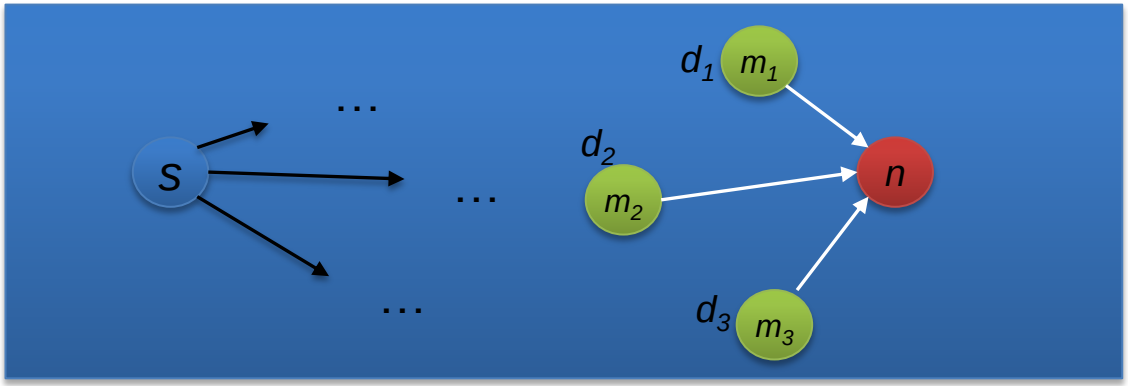

8
1
9
10
0
2
3
9
4
6
7
5
5
2
7

Алгоритм Дейкстры

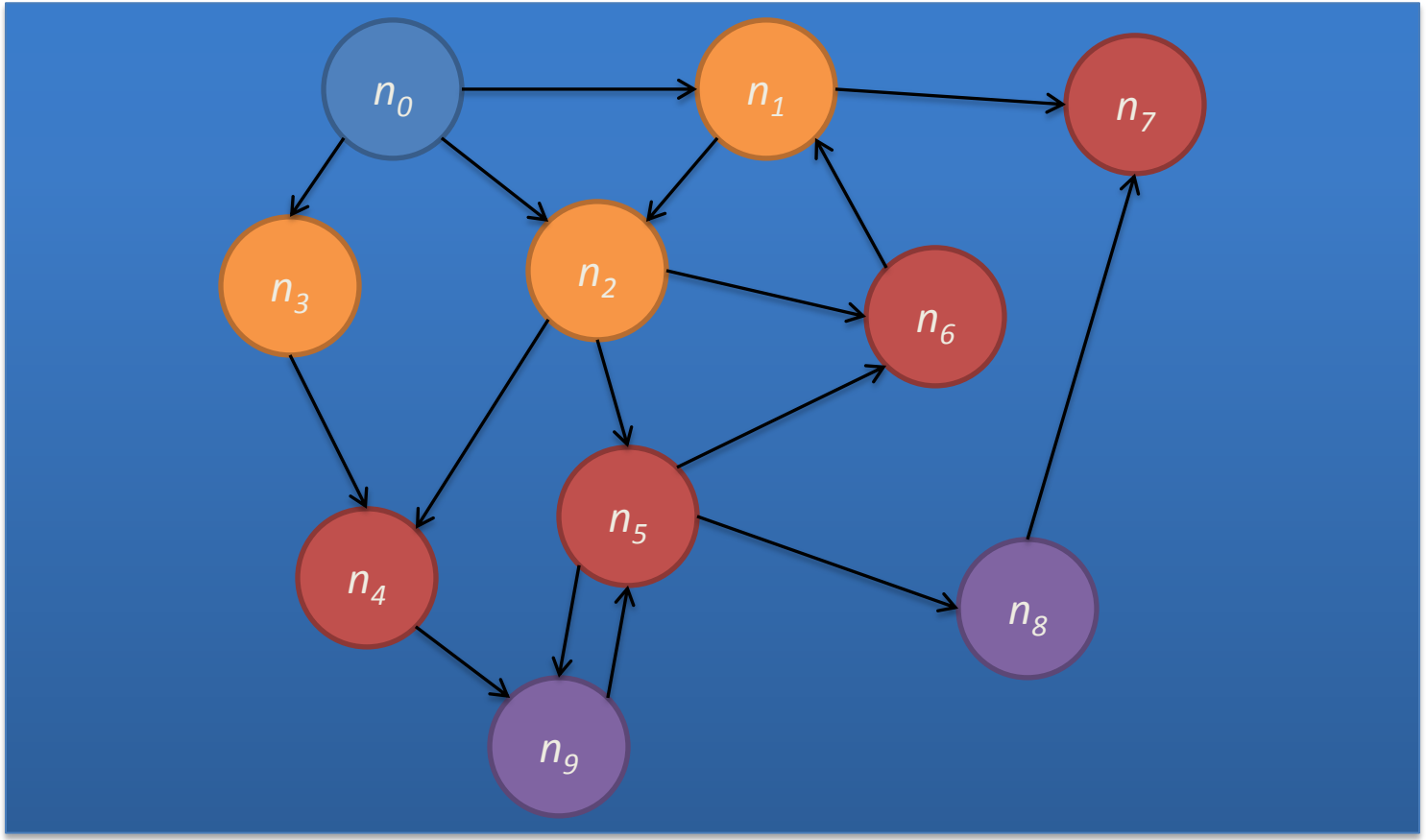
```
Dijkstra(V, s, w)
    for all vertex v ∈ V do
        d[v] ← ∞
    d[s] ← 0
    Q ← {V }
    while Q != Ø do
        u ←ExtractMin(Q)
        for all vertex v ∈ u.AdjacencyList do
            if d[v] > d[u] + w(u, v) then
                d[v] ← d[u] + w(u, v)
```

Поиск кратчайшего пути

- Пусть веса ребер равны 1
- Решение по индукции:
  - *DistanceTo(s) = 0*
  - *DistanceTo(s->p) = 1*
  - *DistanceTo(n) = 1 + min(DistanceTo(m), m ∈ M)*

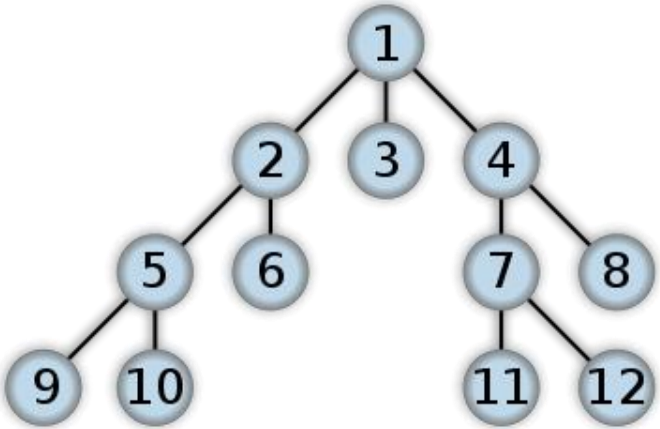


Параллельный поиск в ширину (BFS)



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)
  ∀m ∈ value.adjacency_list: emit (m, value.d + 1)
```

Breadth First Search: Mapper

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

Breadth First Search: Reducer

Reduce In:

```
2 -> {[1, {} ], [∞, {5, 6}]}
```

Reduce Out:

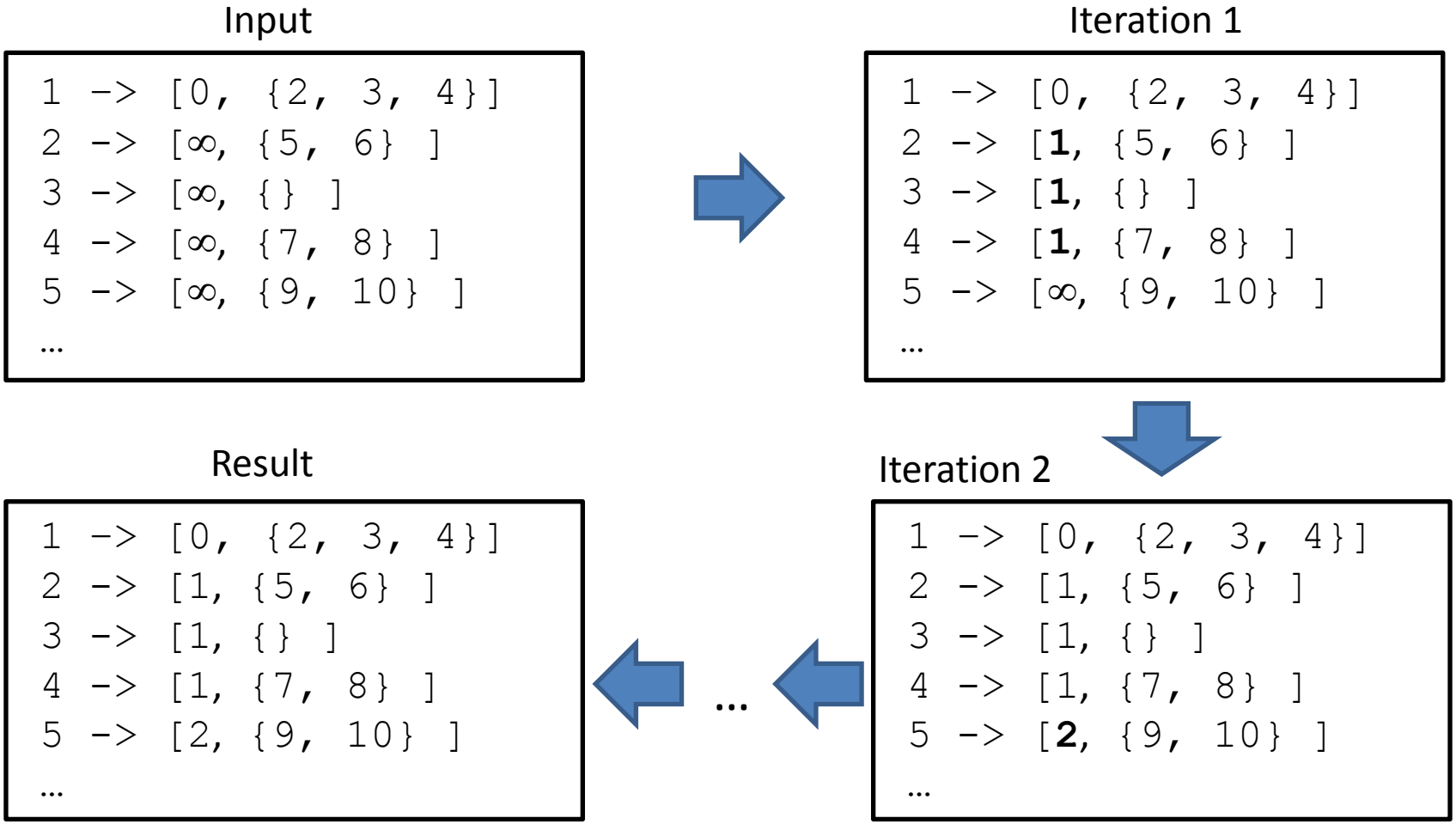
```
2 -> [1, {5, 6} ]
```

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
```

BFS: псевдокод

```
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
                dmin ← d
        M.Distance ← dmin   // Update shortest distance
        Emit(nid m, node M)
```


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

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
- 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

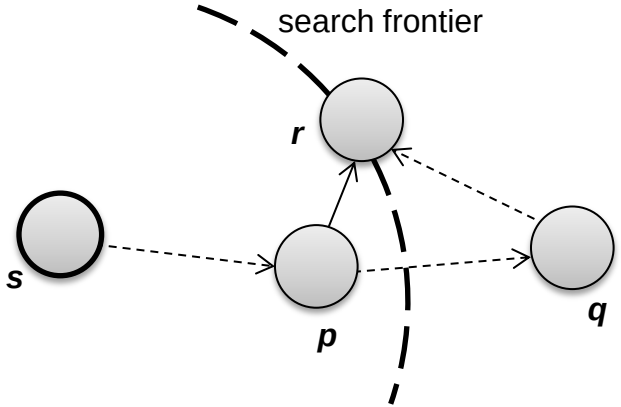
BFS: Weighted Edges

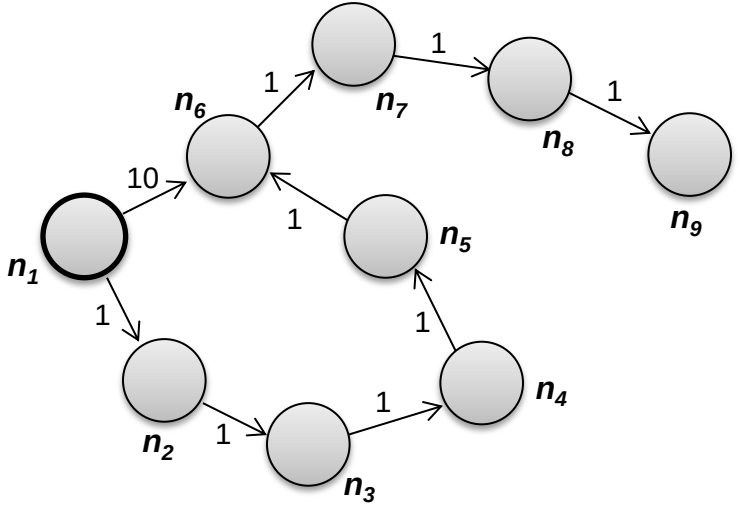
- Добавим положительный вес каждому ребру
- Простая доработка: добавим вес *w* для каждого ребра в список смежных вершин
  - В mapper, emit (m, $d + w_p$) вместо (m, $d + 1$) для каждой вершины m

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





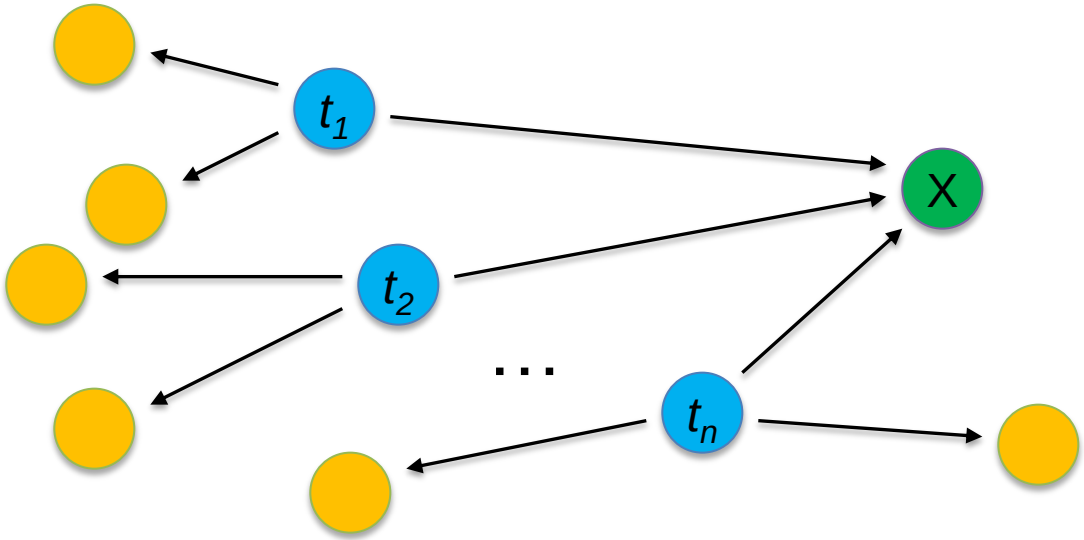
BFS Weighted: критерий завершения

- 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

PageRank

PageRank

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



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

Вычисление PageRank

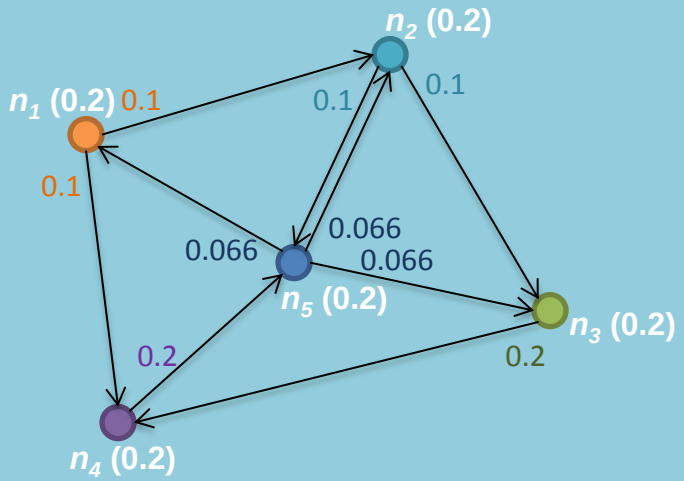
- Свойства PageRank’а
  - Может быть рассчитан итеративно
  - Локальный эффект на каждой итерации
- Набросок алгоритма
  - Начать с некоторыми заданными значения PR_i
  - Каждая страница распределяет PR_i “кредит” всем страниц, на которые с нее есть ссылки
  - Каждая страница добавляет весь полученный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

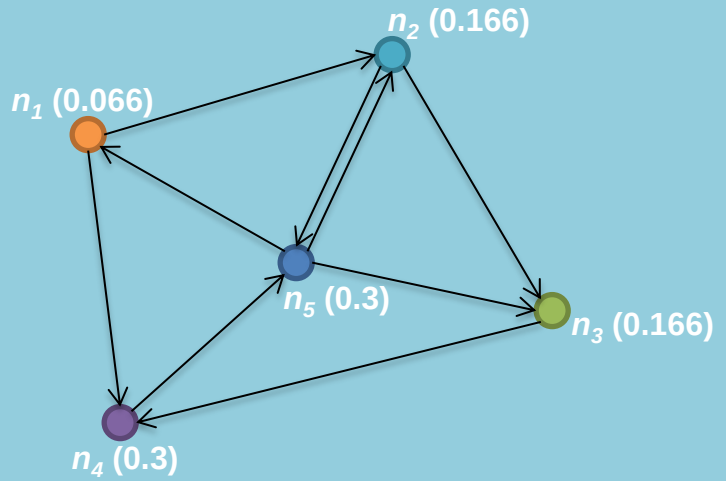
Упрощения для PageRank

Рассмотрим простой случай

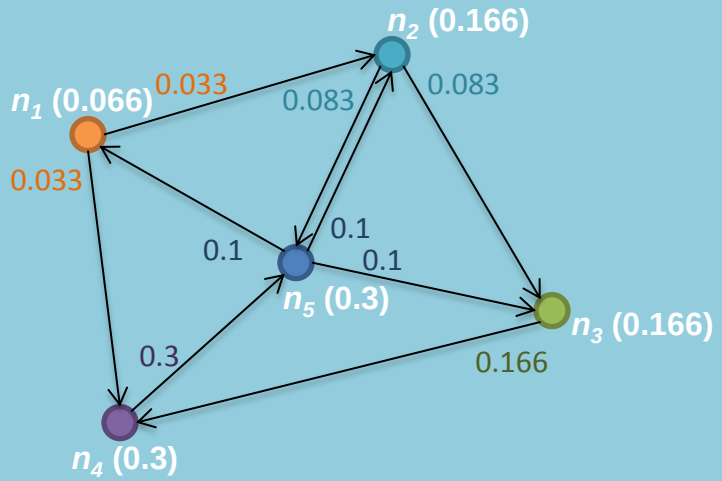
- Нет фактора случайного перехода (*random jump*)
- Нет “подвисших” вершин

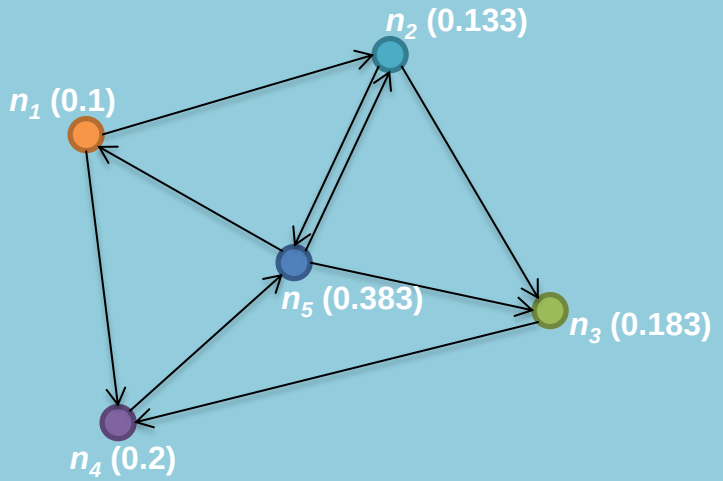
Iteration 1

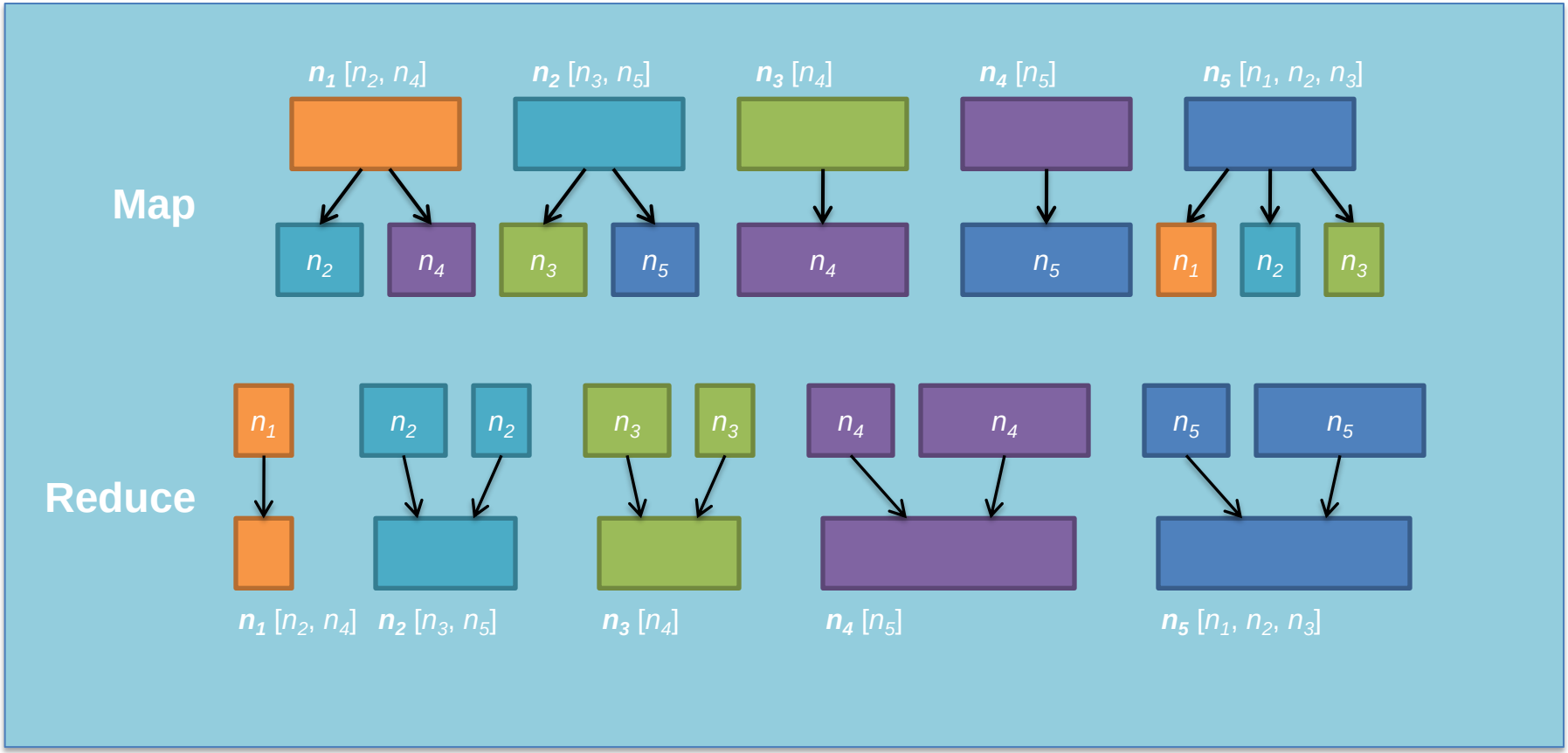

n_2 (0.2)
n_1 (0.2)
0.1
0.1
0.1
0.1
0.066
0.066
0.066
n_5 (0.2)
n_3 (0.2)
0.2
0.2
n_4 (0.2)


n_2 (0.166)
n_1 (0.066)
n_5 (0.3)
n_3 (0.166)
n_4 (0.3)

Iteration 2


n_2 (0.166)
n_1 (0.066)
0.033
0.083
0.083
0.033
0.1
0.1
0.1
n_5 (0.3)
n_3 (0.166)
0.3
0.166
n_4 (0.3)


n_2 (0.133)
n_1 (0.1)
n_5 (0.383)
n_3 (0.183)
n_4 (0.2)


Map
n1 [n2, n4]
n2 [n3, n5]
n3 [n4]
n4 [n5]
n5 [n1, n2, n3]
n2
n4
n3
n5
n4
n5
n1
n2
n3
Reduce
n1
n2
n2
n3
n3
n4
n4
n5
n5
n1 [n2, n4]
n2 [n3, n5]
n3 [n4]
n4 [n5]
n5 [n1, n2, n3]

PageRank: Mapper

```
class Mapper
      method Map(nid n, node N)
          p ← N.PageRank/|N.AdjacencyList|
          Emit(nid n,N)
          for all nodeid m ∈ N.AdjacencyList do
              Emit(nid m, p)
```

PageRank: Reducer

```
class Reducer
      method Reduce(nid m, [p1, p2, . . .])
         M ← ∅
         for all p ∈ counts [p1, p2, . . .] do
            if IsNode(p) then
               M ← p
            else
               s ← s + p
         M.PageRank ← s
         Emit(nid m, node M)
```

Полный PageRank

- Обработка “подвешенных” вершин
- Случайный переход (*random jump*)

$$p' = \alpha \left(\frac{1}{N} \right) + (1 - \alpha) \left(\frac{m}{N} + p \right)$$

Сходимость PageRank

- Продолжать итерации пока **значения** PageRank не перестанут изменяться
- Продолжать итерации пока **отношения** PageRank не перестанут изменяться
- Фиксированное число итераций

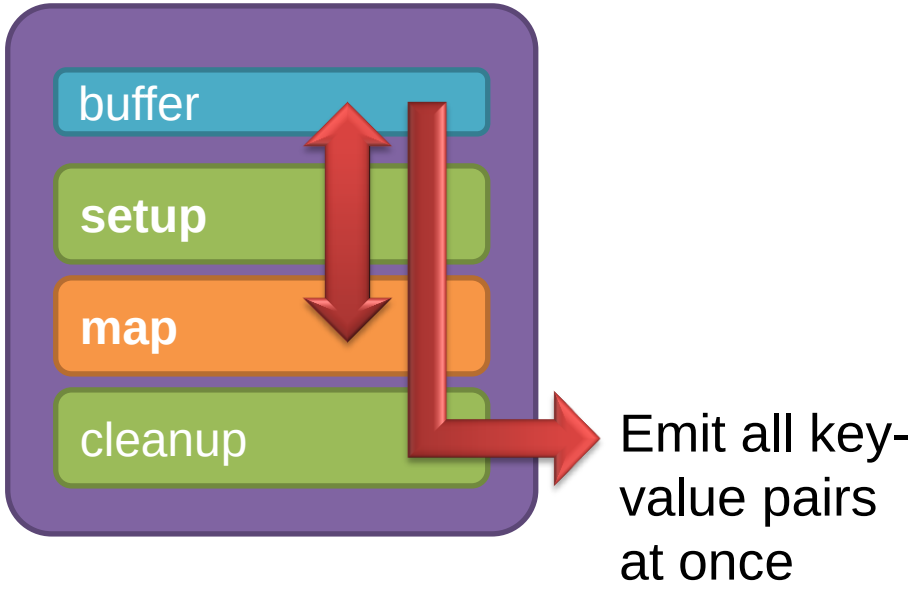
Проблемы MapReduce на графах

MapReduce на графах, проблемы

- 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
  - Агрегирует данные на mapper
  - Но, промежуточные данные все равно обрабатываются
- In-mapper combining
  - Агрегируем сообщения в буффере
  - Но, требуется управление памятью

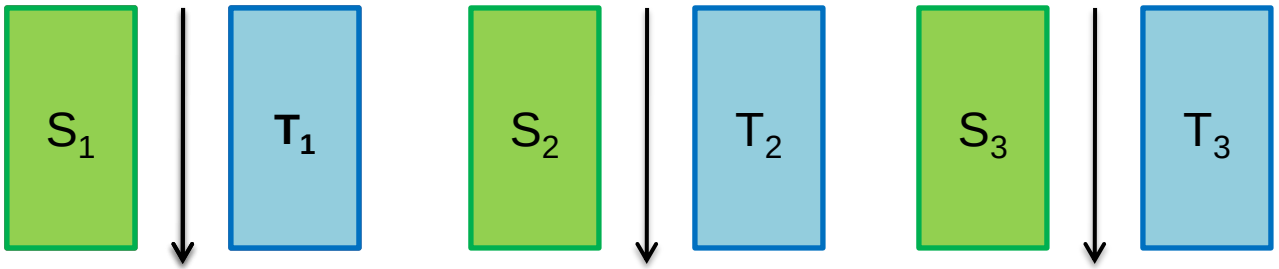


Улучшение партиционирования

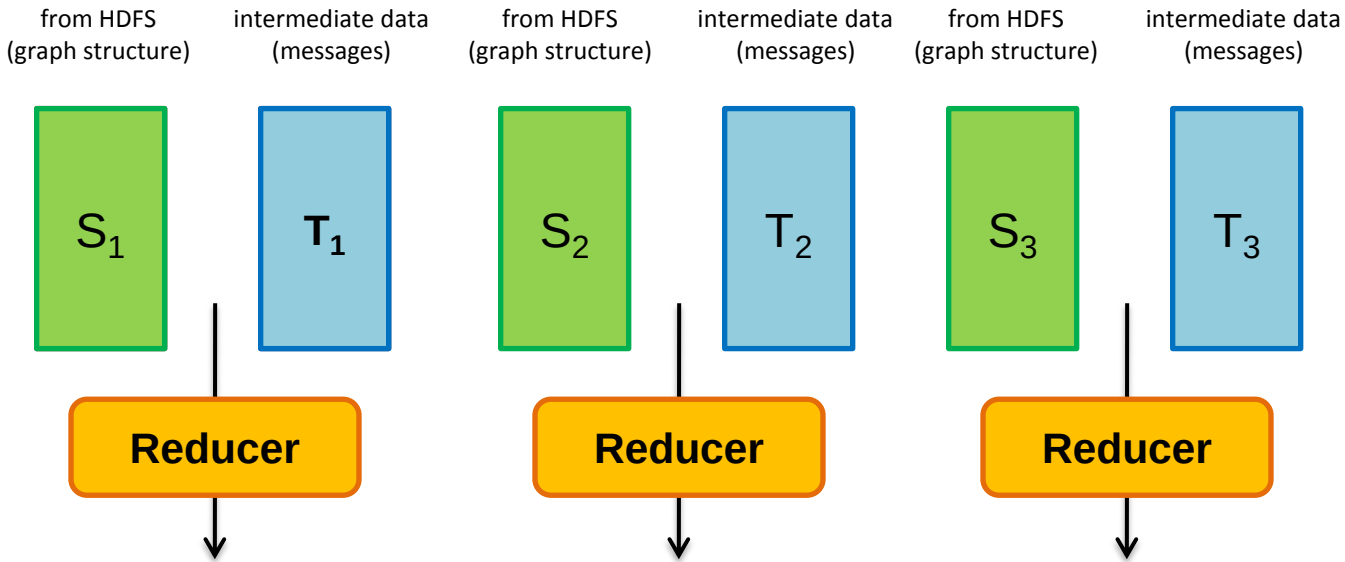
- По-умолчанию: hash partitioning
- Наблюдение: много графов имеют локальную структуру
  - Например, коммьюнити в соц.сетях
  - Улучшение локальной агрегации
- Но, партиционирование довольно **сложно**!
  - Иногда простые эвристики помогают
  - Для веб-графа: использовать партиционирование на основе домена от URL

Schimmy Design Pattern

- Обычно два набора данных:
  - *Messages* (актуальные вычисления)
  - *Graph structure* (структура обрабатываемого графа)
- Schimmy: выполнять shuffle только для *messages*

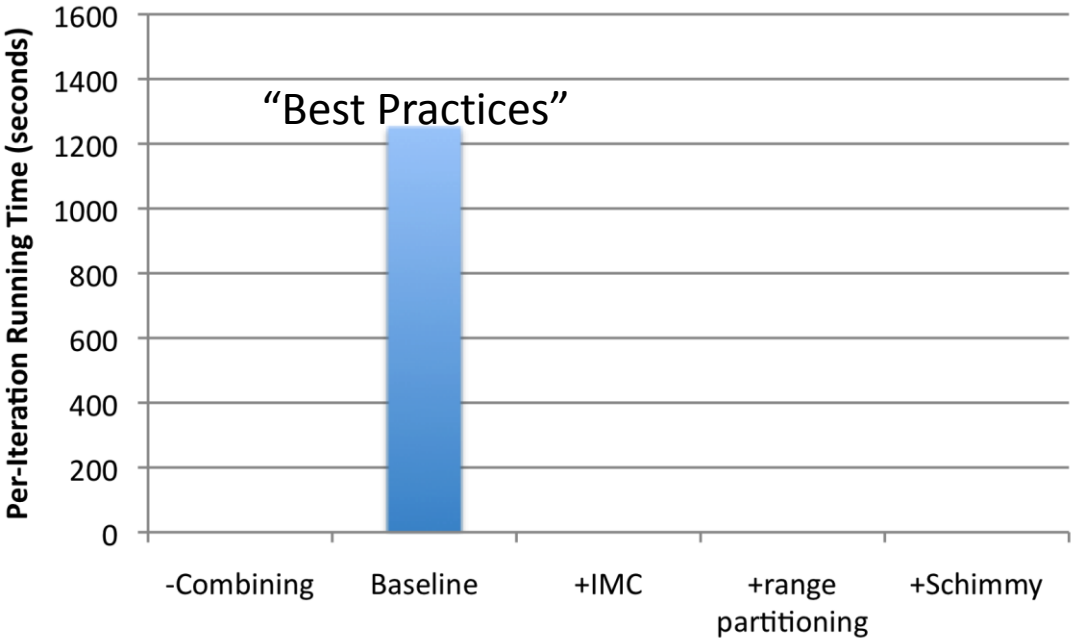


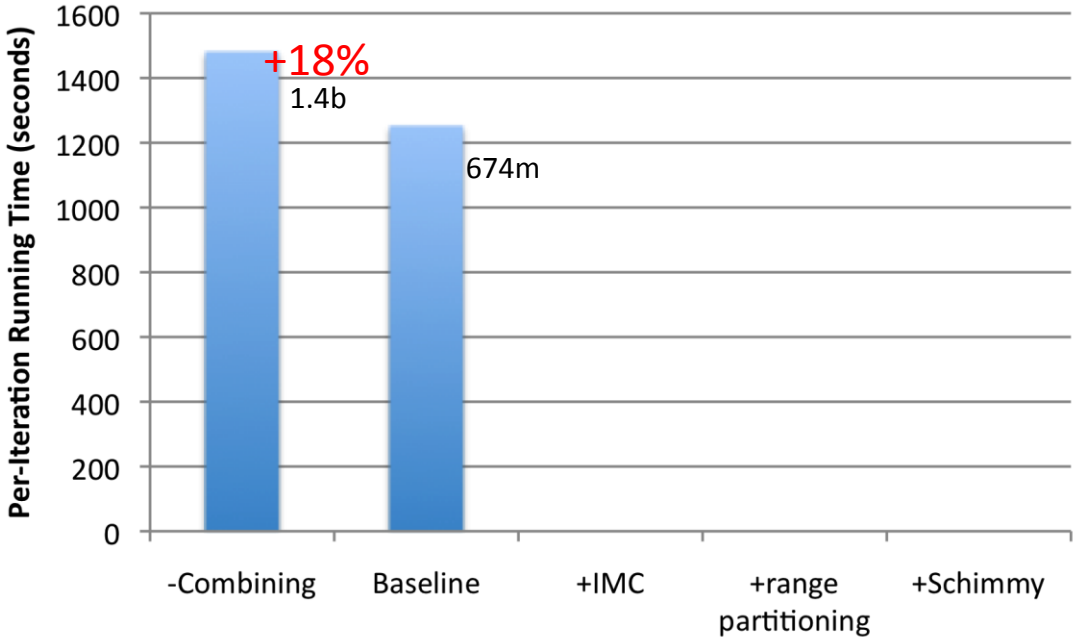
Обе части (S и T) консистентно партиционированы и сортированы по join key

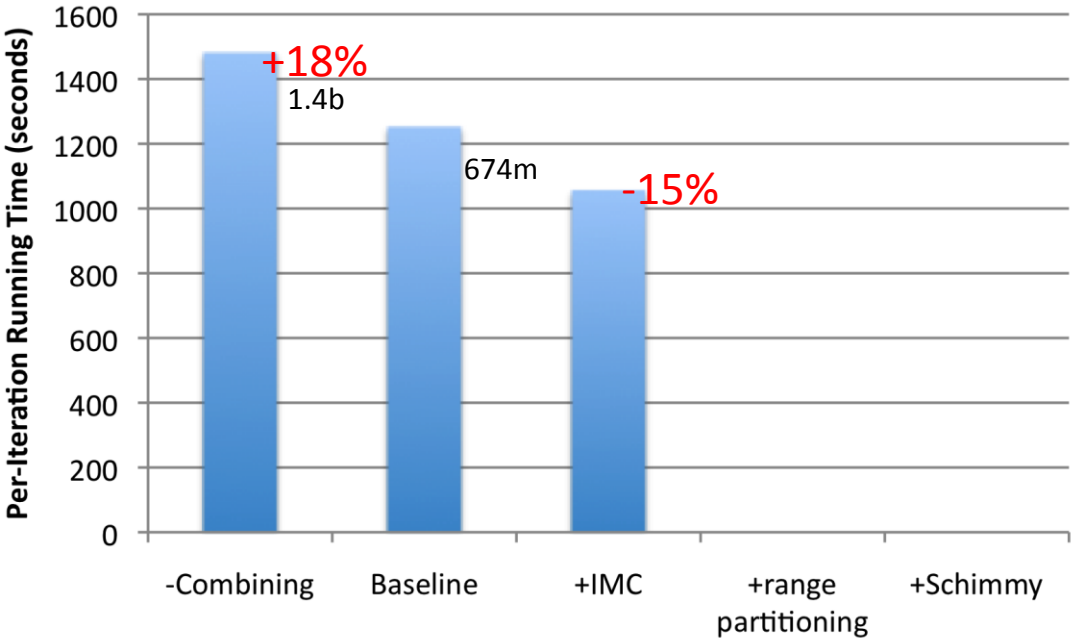


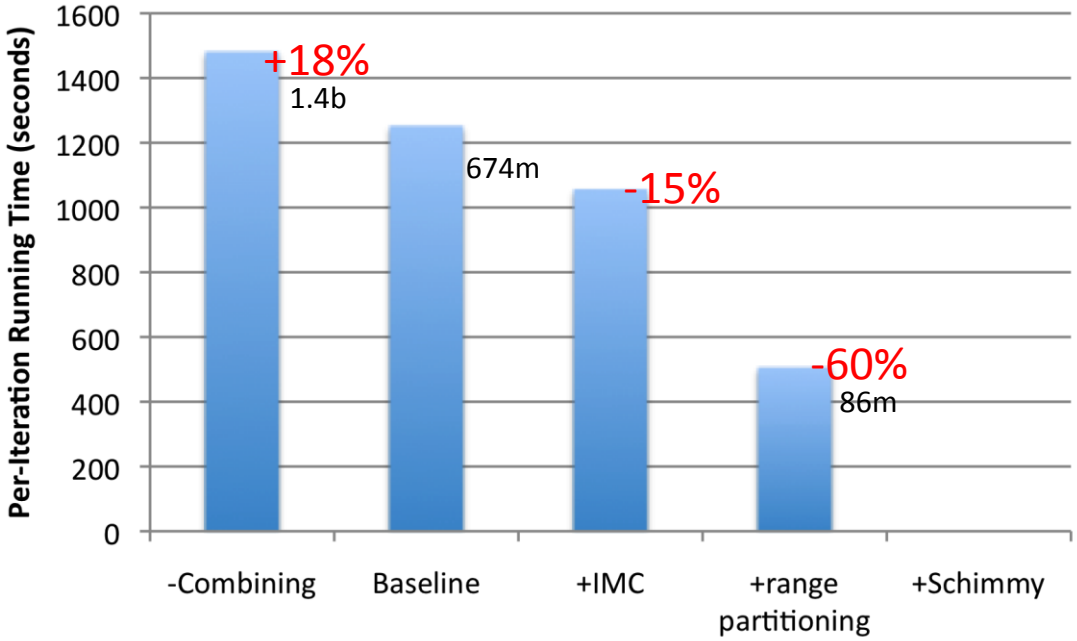
Эксперимент

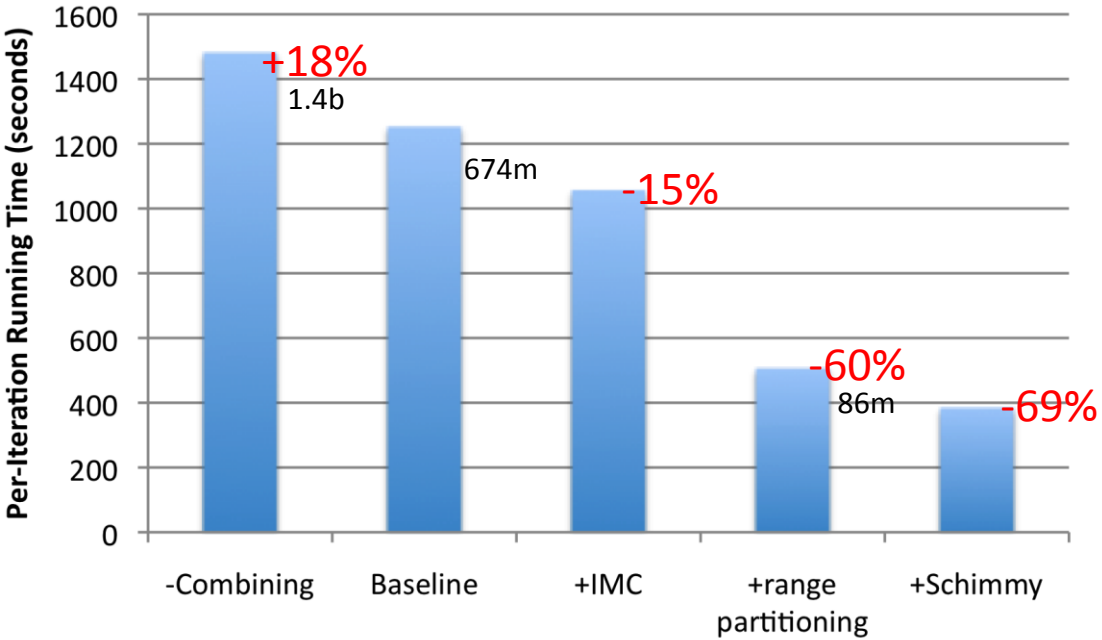
- Cluster setup:
  - 10 workers, each 2 cores (3.2 GHz Xeon), 4GB RAM, 367 GB disk
  - Hadoop 0.20.0 on RHELS 5.3
- Dataset:
  - Первый сегмент английского текста из коллекции ClueWeb09
  - 50.2m web pages (1.53 TB uncompressed, 247 GB compressed)
  - Extracted webgraph: 1.4 Млрд ссылок, 7.0 GB
  - Dataset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


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


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


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


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


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