Lecture 04.02

Map-Reduce

Algorithms

By Marina Barsky
Winter 2017, University of Toronto

Example 1: Language Model

- **Statistical machine translation:**
    - Need to count number of times every 5-word sequence occurs in a large corpus of documents

- **Very easy with MapReduce:**
    - **Map:**
        - Extract (5-word sequence, count) from document
    - **Reduce:**
        - Combine the counts

Example 2. Integers

- Design MapReduce algorithms to take a very large file of integers and produce as output:

(a) The largest integer.

(c) The same set of integers, but with each integer appearing only once.

(d) The count of the number of distinct integers in the input.

Max integer

```
map (file_id, Iterator numbers)
    max_local: = MIN_INTEGER
    for each number n in numbers
        if (n > max_local)
            max_local: = n
    emit_intermediate ("max", max_local)

reduce (single_key, Iterator all_maxes)
    max_total: = MIN_INTEGER
    for each number n in all_maxes
        if (n > max_total)
            max_total : = n
    emit ("max_total" , max_total)
```

Example 3: Inverted index

- Each document has a unique document ID
- Forward index:
  - Given doc ID – retrieve document content
- Inverted index:
  - From document content to document ID
  - Similar (to secondary indexes) idea from information - retrieval community, but:
    - Record → document.
    - Search key → presence of a word in a document.

Inverted index for tweeter

- Input:
  - (tweet1, “I love pancakes for breakfast”)
  - (tweet2, “I dislike pancakes”)
  - (tweet3, “What should I eat for breakfast?”)
  - (tweet4, “I love to eat”)

- Output:
  - (“pancakes”, [twet1, tweet2])
  - (“breakfast”, [tweet1, tweet3])
  - (“eat”, [tweet3, tweet4])
  - (“love”, [tweet1, tweet4])

Inverted index

Input: distributed file with lines (tweet_id, tweet_body)

```
map (input_key, input_value)
    for each line in input_value
        tokens: = split (line)
        tweet_id: = tokens[0]
        tweet_body: = tokens[1]
        for each word in tweet_body
            emit_intermediate (word, tweet_id)

reduce (word, Iterator tweet_ids)
```

Reduce is empty

Example 4: social network analysis

- Input:

$\overrightarrow{\text{Jim, Sue}}$

Jim, Linn

Linn, Joe

Joe, Linn

Kai, Jim

Jim, Kai

- Output 1 Following (count):
- Jim, 3
- Sue, 0
- Linn, 1
- Joe, 1
- Kai, 1

- Output 2 Followers (count):
- Jim, 1
- Sue, 1
- Linn, 2
- Joe, 1
- Kai, 1

- Output 3 Friends (count):
- Jim, 1
- Sue, 0
- Linn, 1
- Joe, 1
- Kai, 1

Followers: list of followers for each user

```
map (file_name, edges)
    for each edge in edges
        emit_intermediate (edge[1], edge[0])

reduce (user_id, Iterator followers)
```

Example 5: Set projection $S=\pi_{a1..an}(R)$

- The ***map*** function:

For each tuple *t* in *R*, construct a tuple *t'* by eliminating from *t* those components whose attributes are not in *S*. Output the key-value pair (*t'*, *t'*).

- The ***reduce*** function:

For each key *t'* produced by any of the *map* tasks, there will be one or more key-value pairs (*t'*, *t'*). Reduce to:

(*t'*, [*t'*, *t'*, ... ,*t'*]) -> (*t'*, *t'*)

- The duplicate elimination is associative and commutative, so a ***combiner*** associated with each map task can eliminate whatever duplicates are produced locally.
- However, the *reduce* tasks are still needed to eliminate two identical tuples coming from different *map* tasks.

Duplicate elimination

```
map (file_id, Iterator numbers)
    for each number n in numbers
        emit_intermediate (n, 1)

reduce (unique_number, Iterator all_occurrences)
        emit (unique_number , unique_number)
```

Example 6: Join

- **Task: compute natural join** *R(a,b)* ⋈ *S(b,c)*

- **Use a hash function *h* from *b*-values to *1...R***
- A **Map** process turns:
  - Each input tuple *R(a,b)* into key-value pair *(b,(a,R))*
  - Each input tuple *S(b,c)* into *(b,(c,S))*

- **Map** processes send each key-value pair with key *b* to Reduce process *h(b)*
- Each **Reduce** process matches all the pairs *(b,(a,R))* with all *(b,(c,S))* and outputs *(a,b,c)*.

Join: step-by-step

Employee

Name	SIN
Mary	**111111**
John	**333333**

AssignedDepartment

SIN	Department
111111	**Accounting**
111111	**Sales**
333333	**Marketing**

Employee ⋈ AssignedDepartment

Name	SIN	Department
Mary	111111	Accounting
Mary	111111	Sales
John	333333	Marketing

Join in map reduce: before *map*

Employee

Name	SIN
Mary	**111111**
John	**333333**

AssignedDepartment

SIN	Department
111111	**Accounting**
111111	**Sales**
333333	**Marketing**

$\Rightarrow$

Employee ⋈ AssignedDepartment

Name	SIN	Department
Mary	111111	Accounting
Mary	111111	Sales
John	333333	Marketing

- Join is a binary operation, map reduce is unary (takes a single dataset as input)
- Idea: treat all the tuples together as a single dataset

Additional label to specify relation name

- Single dataset **tuples**:

(111111, 'Employee', 'Mary')

(333333, 'Employee', 'John')

(111111, 'AssignDepartment', 'Accounting')

(111111, 'AssignDepartment', 'Sales')

(333333, 'AssignDepartment', 'Marketing')

Join in map reduce: *map*

(111111, Employee, Mary)
(333333, Employee, John)
(111111, AssignDepartment, Accounting)
(111111, AssignDepartment, Sales)
(333333, AssignDepartment, Marketing)

- For each record in tuples emit key-value pairs
  (111111, (111111, Employee, Mary))
  (333333, (333333, Employee, John))
  (111111, (111111, AssignDepartment, Accounting))
  (111111, (111111, AssignDepartment, Sales))
  (333333, (333333, AssignDepartment, Marketing))

Why do we use SIN as a key?

Join in map reduce: magic shuffle phase

- Everything with the same key is lumped together on a single reducer

(111111, [(111111, Employee, Mary), (111111, AssignDepartment, Accounting), (111111, AssignDepartment, Sales)])

(333333, [(333333, Employee, John), (333333, AssignDepartment, Marketing)]

Join in map reduce: ***reduce***

(111111, [(111111, Employee, Mary), (111111, AssignDepartment, Accounting), (111111, AssignDepartment, Sales)])

(333333, [(333333, Employee, John), (333333, AssignDepartment, Marketing)]

- Applies reduce to a single key-value pair
- Will produce join between values from different relations for a single key
- Locally, inside each key-list pair – full cross-product

Join in map reduce

```
map (relation_name, (join_attr_name, relation))
    for each tuple t in relation
        emit_intermediate (t[join_attr_name], (relation_name, t))

reduce (join_attr_val, Iterator tuples_to_join)
    emp_tuples: = []
    dept_tuples: = []
    for each v in tuples_to_join
        if (v [0] = “Employee”)
           emp_tuples += v[1]
        else
           dept_tuples += v[1]
    for each e in emp_tuples
        for each d in dept_tuples
            emit (join_attr_val , (e,d))
```

Example 7: PageRank and matrix-vector multiplication

- Originally, map-reduce was designed for fast computation of web page ranks using *PageRank* algorithm

How to rank web pages

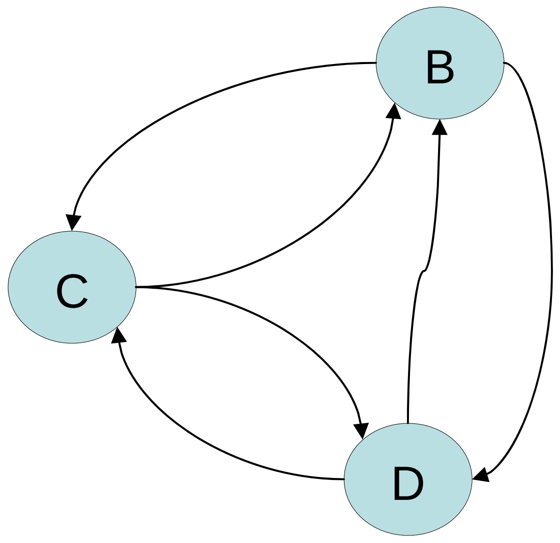
Definition: A webpage is important if many important pages link to it.

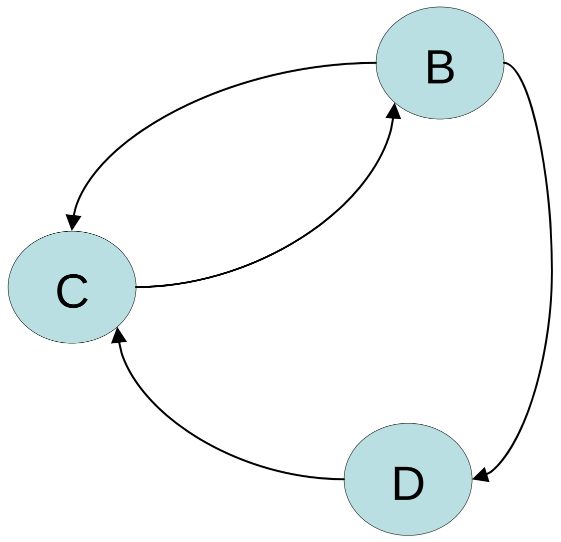
It seems that:

- a problem is the self-referential nature of this definition
- if we follow this line of reasoning, we might find that the importance of a web page depends on itself!

Modeling the web

What can we speculate about the relative importance of pages in each of these graphs, solely from the structure of the links (which is anyways the only information at hand)?



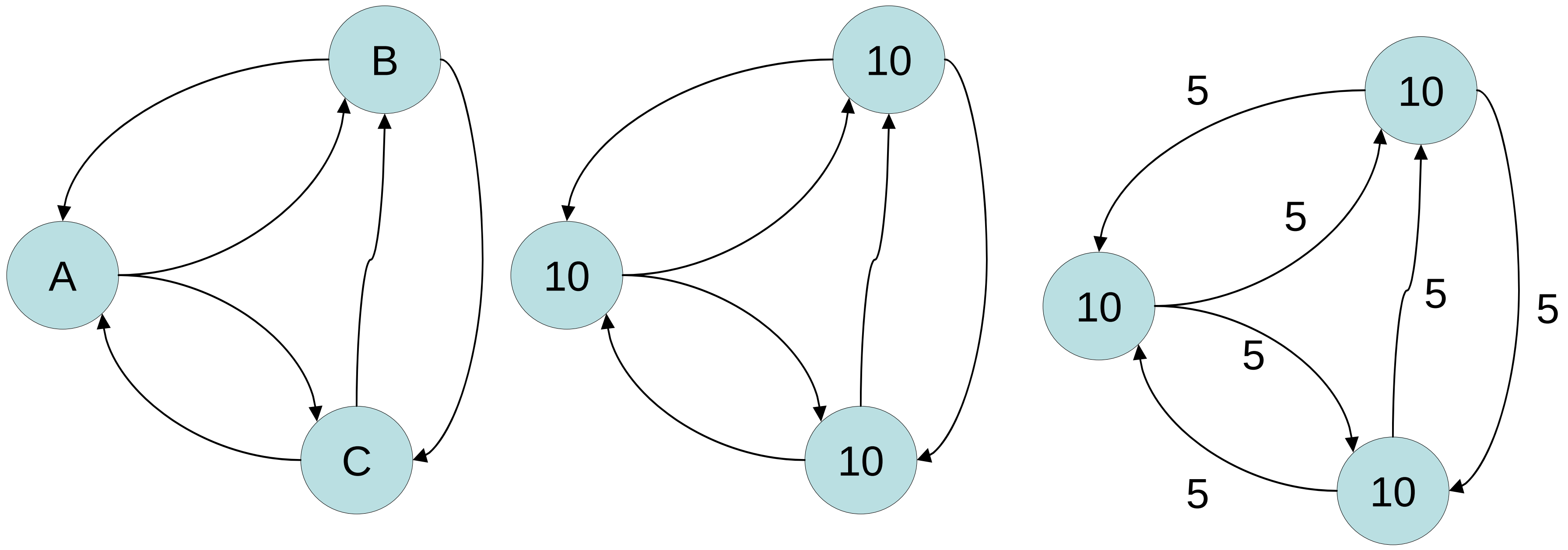


Model: traffic and mindless surfing

- Assumptions:
  - The WEB site is important if it *gets a lot of traffic*.
  - Let assume that everyone is surfing spending a second on each page and then randomly following one of the available links to a new page.
  - In this scheme it is convenient to make sure a surfer cannot get stuck, so we make the following

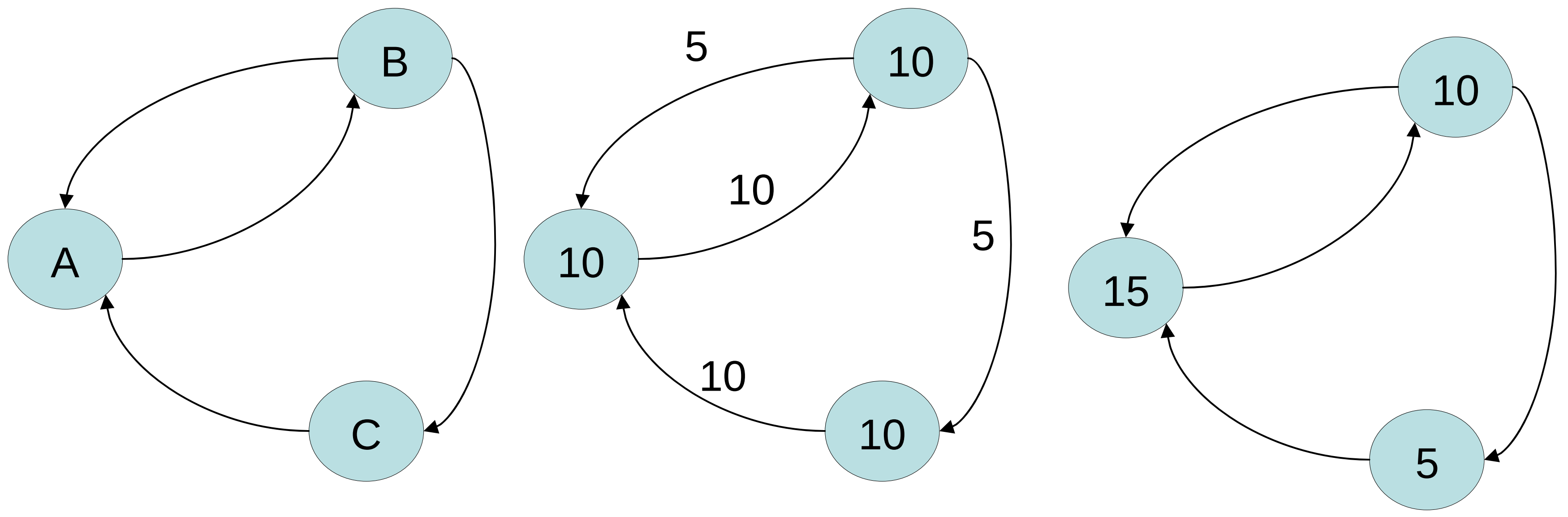
  STANDING ASSUMPTION: Each page has at least one outgoing link.

Stable traffic example



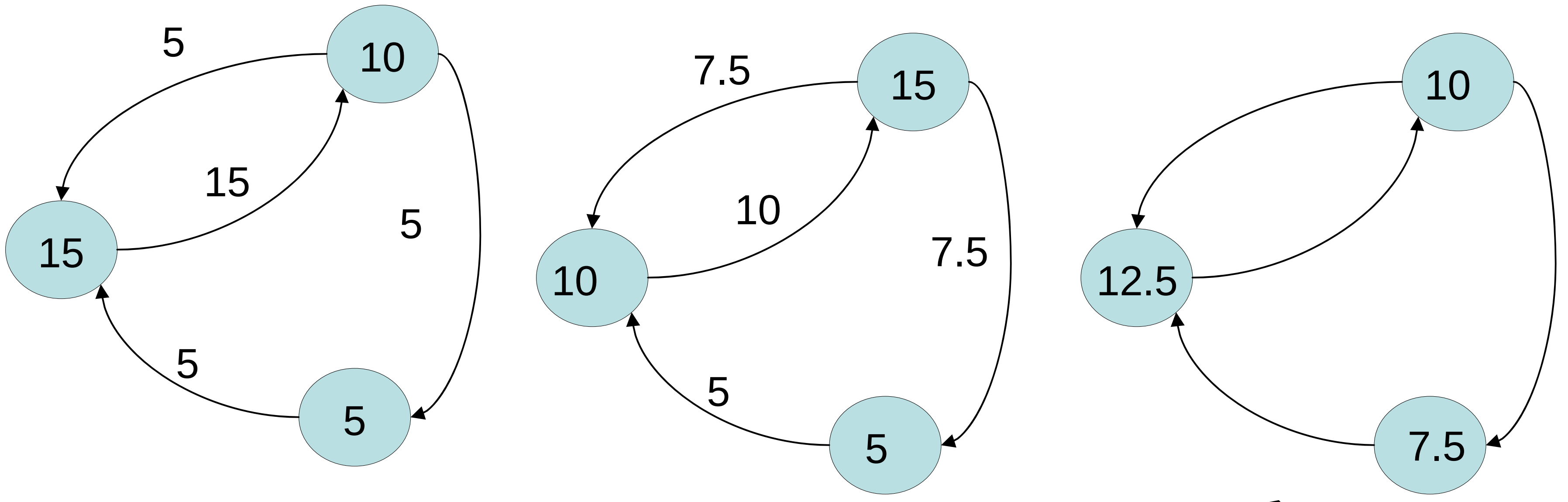
- We start with 10 surfers at each page
- At the first random click, 5 of the surfers at page A, say, go to page B, and the other 5 go to page C. So while each site sees all 10 of its visitors leave, it gets 5 + 5 incoming visitors to replace them:
- **So the amount of traffic at each page remains constant at 10.**

Unstable traffic example



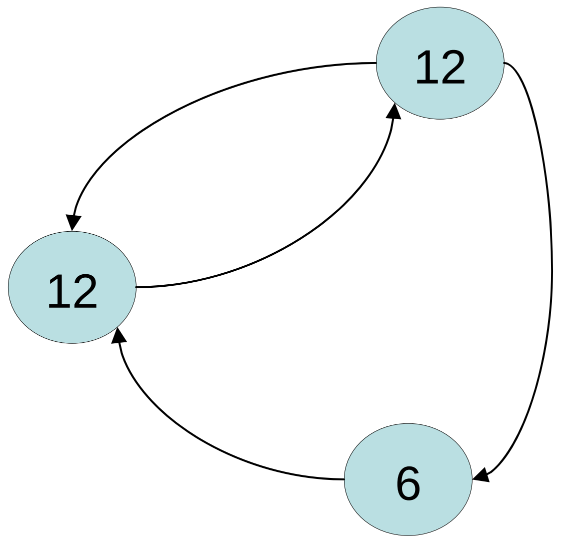
- We start with 10 surfers in each page
- After the first random click, 10 of the surfers at page A go to page B, since there is only 1 outgoing link from A etc…

Unstable traffic example contd.



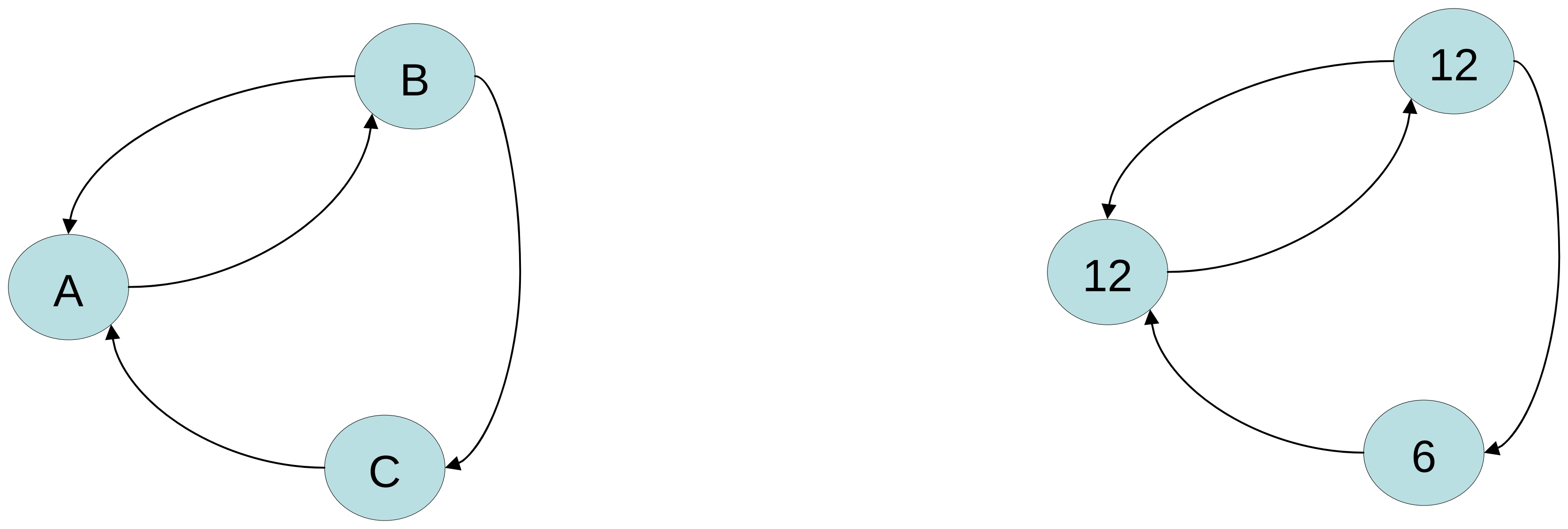
- After the two next iterations it becomes
- Where is this leading? Do we ever reach a stable configuration, as in the first graph?

Traffic converges



- While the answer is no, it turns out that the process **converges** to the following distribution, which you can check oscillates around these values going forward in time
- This stable distribution is what the PageRank algorithm (in its most basic form) uses to assign a rank to each page: The two pages with 12 visitors are equally important, and each more important than the remaining page having 6 visitors.

Question



- How do we qualitatively explain why two of the pages in this model should be ranked equally, even though one has more incoming links than the other?

How to compute the stable distribution?

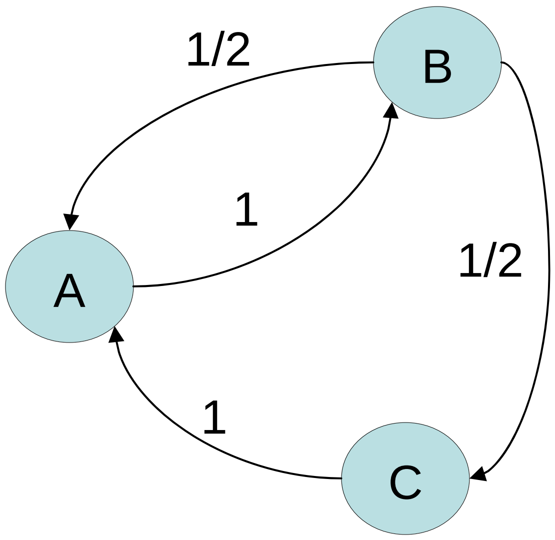
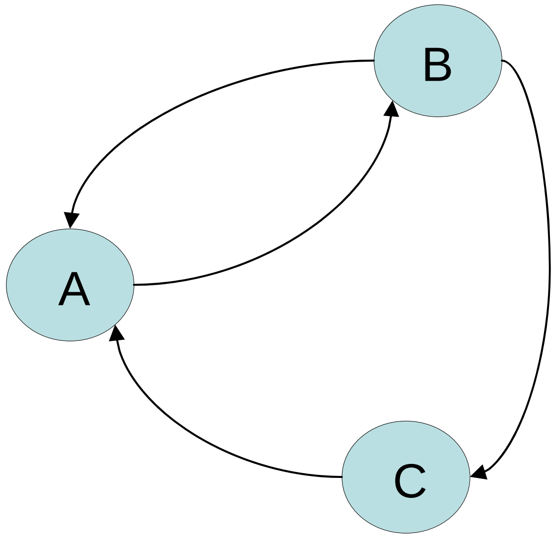


Table of transitions:

transition matrix based on outgoing links

Links from:	A	B	C
A	0	1/2	1
B	1	0	0
C	0	1/2	0

Set initial importance for all pages to 1



Vector of importance

A	1
B	1
C	1

Transition matrix

	A	B	C
A	0	1/2	1
B	1	0	0
C	0	1/2	0

Iteration 1

Current Vector of importance

A	1
B	1
C	1

Transition matrix

	A	B	C
A	0	1/2	1
B	1	0	0
C	0	1/2	0

New Vector of importance

From B

From C

A	1*1/2 + 1*1 =1.5
B	1*1 = 1
C	1*1/2 = 1/2

Find new importance based on number of incoming visitors and their rank

Iteration 2

Current Vector of importance

A	1.5
B	1
C	0.5

Transition matrix

	A	B	C
A	0	1/2	1
B	1	0	0
C	0	1/2	0

New Vector of importance

A	$\overset{\text{From B}}{1*1/2} + \overset{\text{From C}}{0.5*1} = 1$
B	1.5*1 = 1.5
C	1*1/2 = 1/2

Find new importance based on number of incoming visitors and their rank

Iteration 3

Current Vector of importance

A	1
B	1.5
C	0.5

Transition matrix

	A	B	C
A	0	1/2	1
B	1	0	0
C	0	1/2	0

New Vector of importance

From B

From C

A	1.5*1/2 + 0.5*1 =1.25
B	1*1 = 1
C	1.5*1/2 = 0.75

Each entry of the vector is updated based on updated entries for other pages – they get updated together

Matrix-vector multiplication

V_{k-1} - Current Vector of importance

A	1
B	1.5
C	0.5

A - Transition matrix

	A	B	C
A	0	1/2	1
B	1	0	0
C	0	1/2	0

V_k - New Vector of importance

From B

From C

A	1.5*1/2 + 0.5*1 =1.25
B	1*1 = 1
C	1.5*1/2 = 0.75

The new vector at each iteration is the result of matrix-vector multiplication:

$V_k = A * V_{k-1}$

Computing matrix-vector multiplication

$$A\mathbf{x} = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \vdots & \vdots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} a_{11}x_1 + a_{12}x_2 + \dots + a_{1n}x_n \\ a_{21}x_1 + a_{22}x_2 + \dots + a_{2n}x_n \\ \vdots \\ a_{m1}x_1 + a_{m2}x_2 + \dots + a_{mn}x_n \end{bmatrix}$$

- Each entry of a new vector **y** is

$$y_i = \sum_{j=1\dots n} a_{ij} * x_j$$

- In other words, it is a dot product of vector **x** with the corresponding row of matrix *A*

Note that the matrix is **very sparse**: each page has a limited number of outgoing and incoming links compared to the total number of web pages. So we are up to **compute several rounds of multiplication of a very sparse matrix by a very large vector**

Basic matrix-vector multiplication in map-reduce: **input**

- Transition matrix (sparse), stored as tuples of type:

(i, j, A_{ij})

- Current vector of page importance, stored as tuples of type

(i, v_i)

Basic matrix-vector multiplication in map-reduce: **map**

Input: two types of tuples

(i, j, A_{ij})

(i, v_i)

```
map
    for each tuple of type A emit_intermediate (i, (i,j,$A_{ij}$))
    for each tuple of type v
        for j from 1 to n
            emit_intermediate (j, (i, vi))
```

Step-by-step example: **input**

	A	B	C
A	0	1/2	1
B	1	0	0
C	0	1/2	0

A	1
B	1
C	1

- Tuples of type A:

(1,2,1/2)

(1,3,1)

(2,1,1)

(3,2,1/2)

- Tuples of type v:

(1,1)

(2,1)

(3,1)

Step-by-step example: **output of *map***

- Tuples of type A:

(1,2,1/2)

(1,3,1)

(2,1,1)

$(\underset{\text{row}}{3},\underset{\text{col}}{2},\underset{\text{val}}{1/2})$

- Tuples of type v:

(1,1)

(2,1)

$(\underset{\text{row}}{3},\underset{\text{val}}{1})$

- Output of type A:

(1, (1,2,1/2))

(1, (1,3,1))

(2, (2,1,1))

(3, (3,2,1/2))

- Output of type v:

(1, (1,1))

(2, (1,1))

(3, (1,1))

(1, (2,1))

(2, (2,1))

(3, (2,1))

(1, (3,1))

(2, (3,1))

(3, (3,1))

Step-by-step example: **after shuffle**

- Output of type A:

(1, (1,2,1/2))

(1, (1,3,1))

(2, (2,1,1))

(3, (3,2,1/2))

- Output of type v:

(1, (1,1))

(2, (1,1))

(3, (1,1))

(1, (2,1))

(2, (2,1))

(3, (2,1))

(1, (3,1))

(2, (3,1))

(3, (3,1))

- At each reducer:

- (1, [(1,2,1/2), (1,3,1), (1,1), (2,1), (3,1)])

- (2, [(2,1,1), (1,1), (2,1), (3,1)])

- (3, [(3,2,1/2), (1,1), (2,1), (3,1)])

Step-by-step example: **reduce**

- At each reducer:

Multiply non-zero entries of row 1 of A by values of v, sum them up and emit result (1, ½+1)

- (1, [(1,2,**<u>1/2</u>**), (1,3,**1**), (1,1), (2,**<u>1</u>**), (3,**1**)])
- (2, [(2,1,1), (1,1), (2,1), (3,1)])
- (3, [(3,2,1/2), (1,1), (2,1), (3,1)])

Basic matrix-vector multiplication in map-reduce: **reduce**

- The Reduce function simply sums all the values associated with a given row ***i***. The result will be a pair **(i, new v_i)**.

We have a distributed file of new entries of v: finished one iteration of PageRank algorithm

Basic matrix-vector multiplication: limitations

- It seems that the vector of current ranks (which can be very large) is required by all reducers. This may lead to a very costly network traffic

- To overcome this, we can partition vector v and matrix A, and process each partition on a separate reducer, but we may require more iterations of map-reduce

Partitioned Matrix-Vector multiplication: main idea

$$A\mathbf{x} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \vdots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \\ x_n \end{bmatrix} = \begin{bmatrix} \boxed{a_{11}x_1 + a_{12}x_2} + \ldots + a_{1n}x_n \\ a_{21}x_1 + a_{22}x_2 + \ldots + a_{2n}x_n \\ \vdots \\ a_{m1}x_1 + a_{m2}x_2 + \ldots + a_{mn}x_n \end{bmatrix}$$

$$y_1 = \sum_{j=1\ldots k} part\ k$$

$$part\ 1 = \sum_{j=1\ldots2} a_{ij} * v_j$$

$$A\mathbf{x} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \vdots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \\ x_n \end{bmatrix} = \begin{bmatrix} a_{11}x_1 + a_{12}x_2 + \ldots + a_{1n}x_n \\ \boxed{a_{21}x_1 + a_{22}x_2} + \ldots + a_{2n}x_n \\ \vdots \\ a_{m1}x_1 + a_{m2}x_2 + \ldots + a_{mn}x_n \end{bmatrix}$$

- Partition matrix into strips, partition vector into chunks
- Entries i…j of vector v are multiplied only by columns i…j of matrix A
- We can perform these partial multiplications as an additional intermediate step of map-reduce, and sum the results in the final step
- The flexibility of map-reduce is that at each step both input and output are a set of key-value pairs

Cost Measures for Algorithms

In MapReduce we quantify the cost of an algorithm using

1. *Communication cost* = total I/O of all processes
2. *Elapsed communication cost* = max of I/O along any path
3. (*Elapsed*) *computation cost* analogous, but count only max running time of a single process

Note that here the big-O notation is not the most useful (adding more machines is always an option)

Example: Cost Measures

- **For a map-reduce algorithm:**
  - **Communication cost =** input file size + 2 $\times$ (sum of the sizes of all files passed from Map processes to Reduce processes) + the sum of the output sizes of the Reduce processes.
  - **Elapsed communication cost** is the sum of the largest input + output for any map process, plus the same for any reduce process

What Cost Measures Mean

- Either the I/O (communication) or processing (computation) cost dominates
    - Ignore one or the other

- Total cost tells what you pay in rent from your friendly neighborhood cloud

- Elapsed cost is wall-clock time using parallelism

Implementations

- Google
  - Not available outside Google
- **Hadoop**
  - An open-source implementation in Java
  - Uses HDFS for stable storage
  - Download: http://lucene.apache.org/hadoop/
- Aster Data
  - Cluster-optimized SQL Database that also implements MapReduce

Summary

- Learned how to ***scale out*** processing of large inputs
- **Map-reduce** framework allows to implement only 2 functions and the system takes care of distributing computations across multiple machines
- Memory footprint is small. Need to care about the size of intermediate outputs – sending them across network may dominate the cost
- We can perform relational operations in map reduce, if the relations are too big to be processed on a single machine

Map-reduce vs. RDBMS

- RDBMS
  - Declarative query languages
  - Schemas
  - Logical data independence
  - Indexing
  - Algebraic optimization
  - ACID/Transactions
- Map-reduce
  - High scalability
  - Fault-tolerance
  - "One-person deployment"

Spark, Pig, Hive, DryadLINQ – try to incorporate this into map-reduce