Large Scale Graph Algorithms

Nishanth Nair

April 07, 2020

Single Source Shortest Path

Distributed Dijkstra's

- On a single-node computer Dijkstra's algorithm proceeds slowly down the tree one level at a time.
- Once you are past the first level, there will be many nodes whose edges need to be examined, and in the code this happens sequentially.
- The key to Dijkstra's algorithm is the **priority queue** that maintains a globally sorted list of nodes by current distance.
- Limitations of a single-node computer:
  - Single-processor machine can be slow and limited: Dijkstra's algorithm.
- **How can we modify this to work for a huge graph and run the algorithm in parallel?**
  - For maximum parallelism, the maps and reducers need to be stateless.
  - They cannot depend on any data generated within the same job.
  - We cannot control the order in which maps or reducers run.
  - Since MapReduce is stateless (the programming model does not provide a mechanism for exchanging global data), Dijkstra's cannot be distributed using map reduce.
- **Solution:**
  - We adopt a brute-force approach known as **parallel breadth-first search**.

General Approach

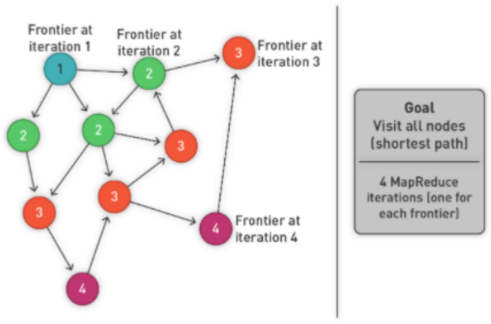


Figure 1: example

- In the distributed approach for BFS, we use multiple iterations of map-reduce
  - In each iteration, the map-reduce task expands the known frontier by 1 hop.
  - Multiple such iterations are required to explore the entire graph
  - After each iteration, the output of the task is fed as input to the next iteration.
  - There will be as many iterations as the number of levels in the graph.
  - Since map-reduce is stateless, the adjacency list is passed around from mapper to reducer.

Parallel BFS (Unweighted)

- Graph is represented as an adjacency list.
- The data needs to be represented as key-value pairs for map-reduce, so we use the following format:
  - **Node, neighbor_list | distance | state**
  - Distance is the distance from source
  - state is the node state. It ccan have the following values
    - **U**: Unvisited (not explored)
    - **Q**: Frontier (Node in queue, yet to be expanded)
    - **V**: Visited (ccaptured node)
- Below is an example of the process:

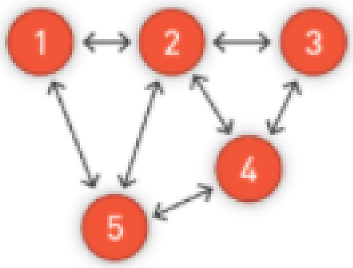
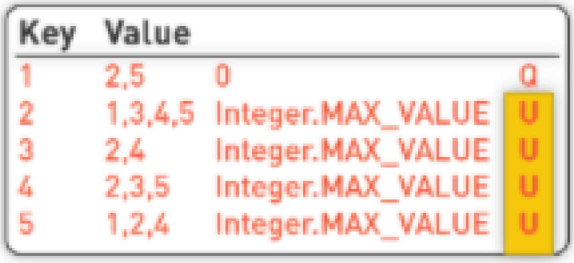


Figure 2: Example Graph

- **STEP 1**: INIT
  - In this step, we initialize the graph and set the source node wwith a distance of 0 and state=Q
  - All other nodes are marked as unvisited (state=U)

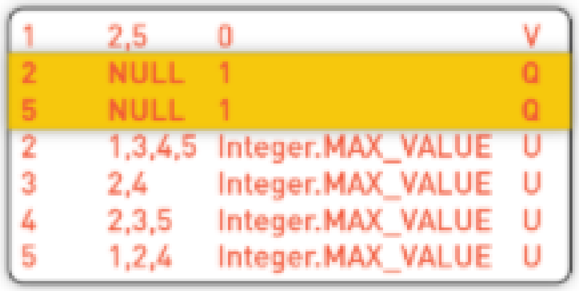


Key	Value		
1	2,5	0	Q
2	1,3,4,5	Integer.MAX_VALUE	U
3	2,4	Integer.MAX_VALUE	U
4	2,3,5	Integer.MAX_VALUE	U
5	1,2,4	Integer.MAX_VALUE	U

Figure 3: Init Step

- **STEP 2**: Mapper
  - Mappers are responsible for expanding the frontier nodes.
  - They do the following based on the node state.
  - If node state = Q,
    - It emits a new node for each neighbor with state = Q and distance = distance+1
      - Since it does not know about the edges, it leaves the neighbor list blank or NULL.

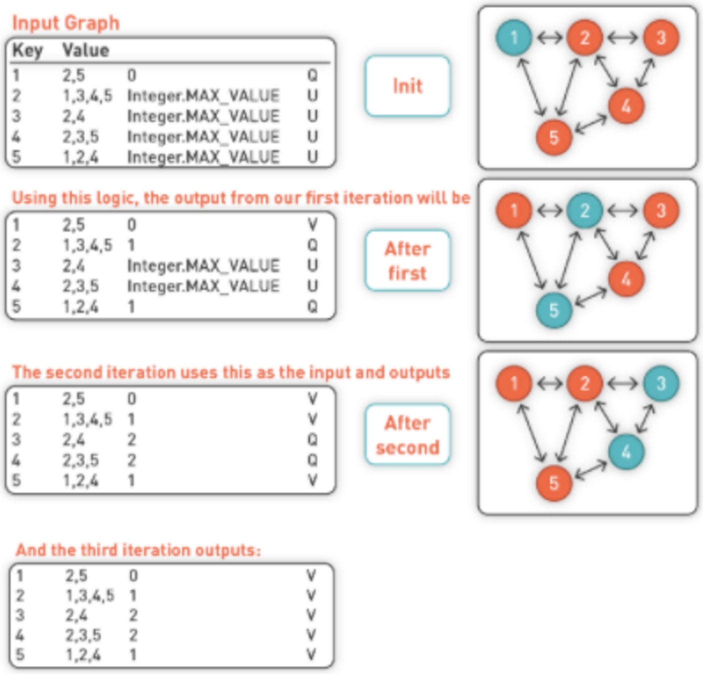
    * Marks the original node with state = V and emits it too.
  - If node state is V or U, it simply emits the node with no changes.
  - So now, we potentiallly have multiple entries per node as the output from a mapper.



1	2,5	0	V
2	NULL	1	Q
5	NULL	1	Q
2	1,3,4,5	Integer.MAX_VALUE	U
3	2,4	Integer.MAX_VALUE	U
4	2,3,5	Integer.MAX_VALUE	U
5	1,2,4	Integer.MAX_VALUE	U

Figure 4: First mapper output

- **STEP 3**: Reducer
  - The reducers receive all the data for a given a node (or key).
  - So, it receives all the “copies” of the given node.
  - From our example, for node 2, it receives
    ```
    2,  NULL    | 1                 | Q
    2,  1,3,4,5 | Integer.max_value | U
    ```
  - The reducer’s job is to take all this data and construct or join a new node using:
    * The nonnull list of edges
    * The minimum distance among values
    * The status
  - The reducer combines the above records and emits a single node as follows:
    ```
    2,  1,3,4,5 | 1 | Q
    ```
- **STEP 4**: Iterate steps 2 and 3 till we traverse entire graph


Input Graph

Key	Value		
1	2,5	0	Q
2	1,3,4,5	Integer.MAX_VALUE	U
3	2,4	Integer.MAX_VALUE	U
4	2,3,5	Integer.MAX_VALUE	U
5	1,2,4	Integer.MAX_VALUE	U

Using this logic, the output from our first iteration will be

1	2,5	0	V
2	1,3,4,5	1	Q
3	2,4	Integer.MAX_VALUE	U
4	2,3,5	Integer.MAX_VALUE	U
5	1,2,4	1	Q

The second iteration uses this as the input and outputs

1	2,5	0	V
2	1,3,4,5	1	V
3	2,4	2	Q
4	2,3,5	2	Q
5	1,2,4	1	V

And the third iteration outputs:

1	2,5	0	V
2	1,3,4,5	1	V
3	2,4	2	V
4	2,3,5	2	V
5	1,2,4	1	V

Figure 5: map-reduce iterations

- **STEP 5**: Termination
  - We are done when there are no output nodes that are frontier (i.e., in Q state).
  - Note: If the graph is not connected,you may have final output nodes that are still unvisited.

Parallel BFS (Weighted)

- The unweighted version doesn't work for weighted graphs as shown in the below example.
- This because, once a node is visited, we assume we already have the shortest path to that node.

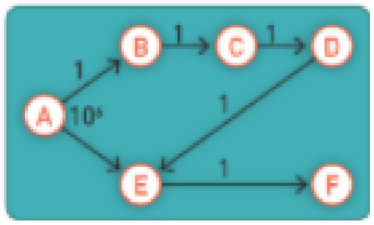


Figure 6: Weighted graph

- In the weighted version, the following changes will be needed:
  - If the distance to a visited node has gotten shorter at later iterations, we need to do the following:

        * Update the distance on the visited node.
        * Update the state to ‘Q’ so that it is re-enqueued.
        * This will result in a recomputation of all nodes emanating from that node.
- In comparison to the above, Dijkstra’s is more efficient.
- However, the map-reduce approach scales well for very large data.
    - Can add more hardware to solve larger problems.
- Passing the graph structure from one iteration to another is very expensive.
    - Special distributed graph libraries like Giraph, Spark Graphx are purpose-built for graph processing to solve the above problem.

Example Mapper Code

```
def mapper(self, _, line):
    path = []
    node,values_str = line.split('\t')
    values = values_str.split('|')

    if len(values) == 4:#Process adjacency list
        edges = ast.literal_eval(values[0])
        distance = int(values[1])
        path = ast.literal_eval(values[2])
        state = values[3]
    else:#Initialize of not already done
        edges = ast.literal_eval(values_str)
        if node == self.startnode:
            state = 'Q'
            distance = 0
        else:
            state = 'U'
            distance = sys.maxint

    if state == 'Q':
        path.append(node)
        yield node,(edges,distance,path,'V')
        if edges is not None:
            for k,v in edges.iteritems():
                yield k,(None,distance+int(v),path,state)
    else:
        yield node,(edges,distance,path,state)
```

Example Reducer Code

```
def reducer(self, key, values):
    edges=None
    ipath={}
    istate=[]
    idistance = {}

    distance = 0
    state=None
    path=[]

    node = key
    for value in values:
        temp_state = value[3]
        istate.append(temp_state)
        idistance[temp_state] = int(value[1])
        ipath[temp_state] = value[2]
        if value[0] != None:
            edges = value[0]

    min_key = min(idistance, key=idistance.get)
    distance = idistance[min_key]
    path = ipath[min_key]

    if 'V' in istate:
        if distance < idistance['V']:#Reset and enqueue
            state = 'Q'
        else:
            state = 'V'
    elif 'Q' in istate:
        state = 'Q'
    else:
        state = 'U'

    yield node,'{0}|{1}|{2}|{3}'.format(edges,distance,path,state)
```

Example Driver Code

```
mr_job = GraphAlgoMRJob(args=[INPUT,
                              '-r',RUN_MODE,
                              '--startnode', START_NODE,
                              '--mappers',MAPPERS,
                              '--reducers',REDUCERS])
counter = 0
while (1):
    STOP = True
    with mr_job.make_runner() as runner:
        counter += 1
        print "iteration␣#{0}".format(counter)
        runner.run()
        with open(INPUT, 'w') as f:
            for line in runner.stream_output():
                f.writelines(line)
                if STOP:
                    key,values_str =  mr_job.parse_output_line(line)
                    values = values_str.split('|')
                    state = values[3].strip()
                    if state in ['Q']:
                        STOP = False

    if(STOP):
        break
```

Web search and graphs

- In a webg graph,
  - The web page is the node
  - The hyperlink to another page is considered the edge.
  - These can stored as an adjacency list.

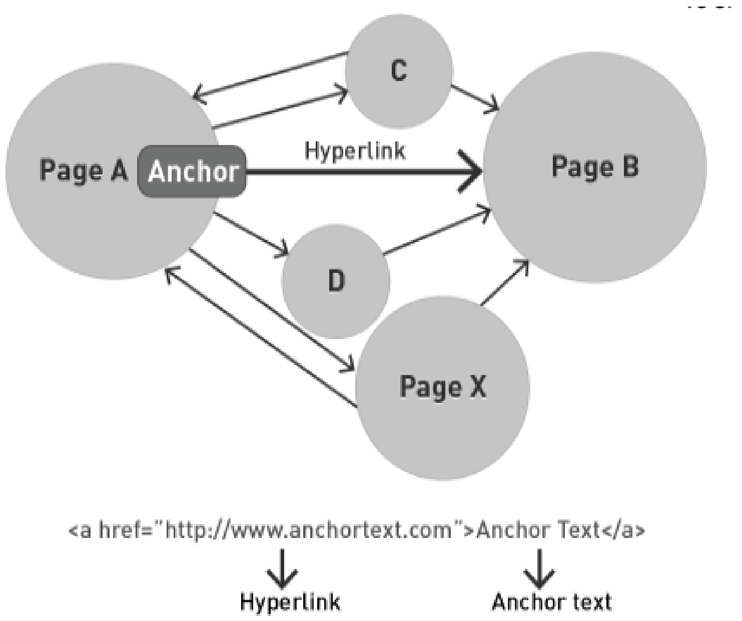


Figure 7: Web graph

- Some initial web search approaches
  - **Link count as a measure of popularity**
    - Each page gets a score
      - Undirected popularity = number of in-links + number of out-links
      - Directed popularity = Number of in-links
    - Reasonable results
    - Issue with spam to game the ratings.
  - **Indexing anchor text**
    - Assumption 1: A hyperlink between pages denotes author-perceived relevance (quality signal)
    - Assumption 2: The text in the anchor of a hyperlink describes the target page (textual context)
    - Approach:
      - When indexing a document D, include anchor text from links pointing to D.
      - Use link count as a measure of static goodness
      - Use anchor text for text match.
    - An unexpected side effect:

- Google bombing and Googlewashing: causing a web page to rank highly in search engine results for unrelated or off-topic search terms by linking heavily

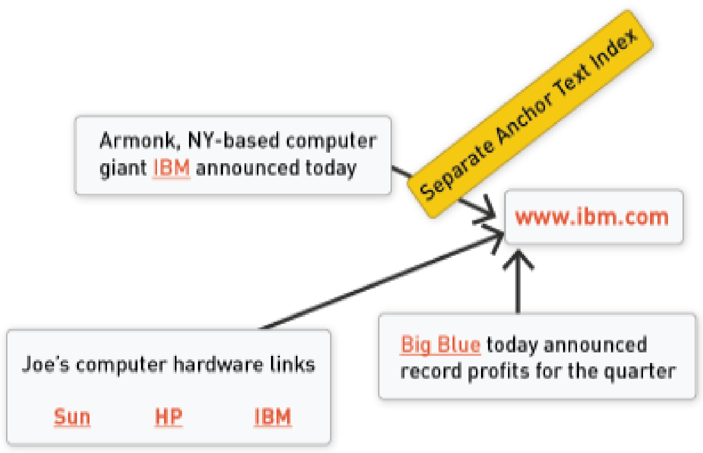

Separate Anchor Text Index
Armonk, NY-based computer giant IBM announced today
www.ibm.com
Joe's computer hardware links
Sun
HP
IBM
Big Blue today announced record profits for the quarter

Figure 8: Indexing anchor text

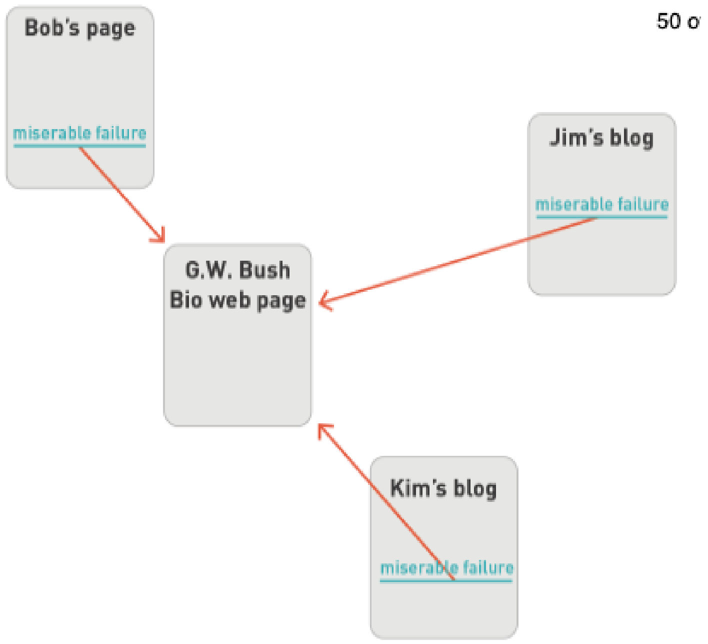


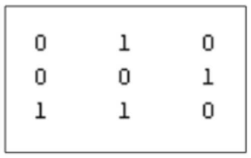
Figure 9: Gaming the anchor index

Page Rank

- PageRank can be thought of as a model of user behavior.
- Assume there is a "random surfer" who is given a web page at random and keeps clicking on links, never hitting "back" but eventually gets bored and starts on another random page.
- **The probability that the random surfer visits a page is its PageRank.**
- Another intuitive justification is that a page can have a high PageRank if there are many pages that point to it, or if there are some pages that point to it and have a high PageRank.
- Intuitively, pages that are well cited from many places around the web are worth looking at.
- Also, pages that have perhaps only one citation from an "important page" are also generally worth looking at.
  - If a page was not high quality, or was a broken link, it is quite likely that an important page would not link to it.

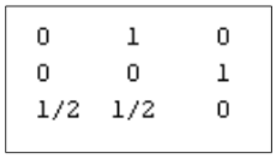
Definitions

- A graph can be reppresented as an **adjacency Matrix** A, where,
  - Weight on edge from i to j is A(i,j)
  - Its symmetric if undirected: A(i,j) = A(j,i)
- A graph can be reppresented as a **Transition matrix** P, where,
  - It is right stochastic matrix, i.e. each row sums to 1.
  - Probability of moving from node i to node j = P(i,j)
  - $P(i,j) = \frac{A(i,j)}{\sum_i A(i,j)}$



0	1	0
0	0	1
1	1	0

Adjacency matrix A



0	1	0
0	0	1
1/2	1/2	0

Transition matrix P

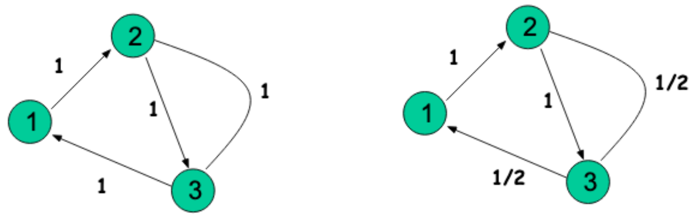


Figure 10: Transition matrix example

- **Random walk**
  - A user starts at a random Web page and either:
    - Randomly clicks on links, surfing from page to page; or
    - Enters a URL in the address bar and then clicks on links.
  - Over time, we mathematically formalize a path of successive random steps.
  - Probability distribution over nodes in the graph represents the likelihood that a random walk over the link structure will arrive at a particular node.
  - PageRank measures the visit rate over an extended period of time (think "infinite time").
- **Probability distribution of random walk**
  - Let the probability that a surfer is at node i at time t is $x_t(i)$
  - Sum of all incoming probabilities to node i is given by:
    - $x_{t+1}(i) = \sum_j$(probability of being at j) $\times$ probability$(j \rightarrow i)$
    - $x_{t+1}(i) = \sum_j x_t(j) \times P(j,i)$, P is the transition matrix of the graph
- **stationary distribution**
  - When a random surfer keeps traversing the graph for a long period of time, the distribution does not change anymore.
  - i.e. $x_{T+1} = x_T$
  - This is called the steady state distribution of the underlying markov process.

Ranking by steady state distribution (Intiution)

- If the webpages are ranked by their steady state distribution defined earlier, we get a form of page rank for these pages.
- This is a form of **Query independent** ranking of the pages.
- If a page is ranked high, it means it gets votes from other highly ranked pages.
- A random surfer would end up visiting these pages more often.
- An alternate interpretation is, simply count the number of times a random surfer would visit a page to create their rankings.

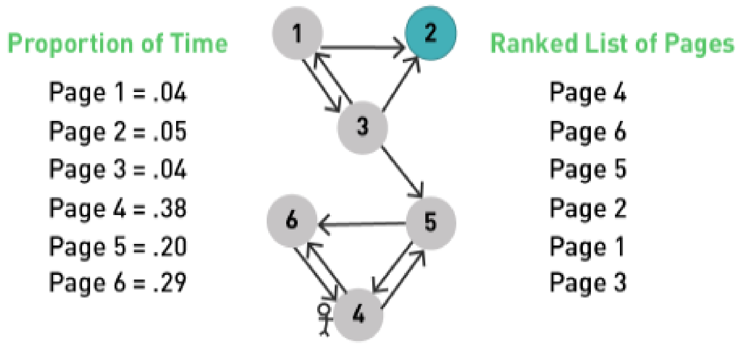


Figure 11: Page ranking by prob distribution

- In the above, node 2 has no out links and is called a **Dangling node**

- The complete formulation of page rank includes a component to handle dangling nodes.

Markov Chain

- A Markov chain is a mathematical system that undergoes transitions from one state to another, between a finite or countable number of possible states.
- Named after Andrey Markov.
- Markov chain is a random process usually characterized as memoryless (i.e follows the markov property)
- **Markov property**:
  - The next state depends only on the current state and not on the sequence of events that preceded it

Components of markov chain

- A markov chain is comprised of 2 components
  - A state vector of size n (i.e n states)
  - A transition probability matrix P of dimension n x n

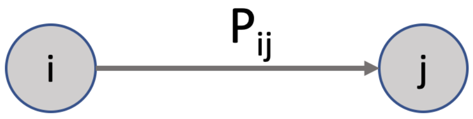


Figure 12: Markov Chain

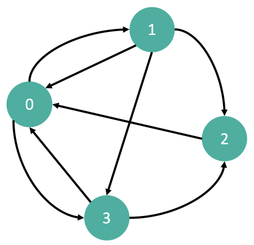
- At each time step, we are exactly in one state.
- For $1 \leq i, j \leq n, P(i, j)$ gives us the probability of j being the next state, given we are currently on state i.
- For each state, $\sum_{j=1}^{n} P(i, j) = 1$
- Markov chains provide a good way to approximate noisy systems:
  - $S_{t+1} = f(S_t, noise)$
  - Tells us about future state.
  - Tells us about few time steps into the future irrespective of starting state.

Stochastic Matrix

- A stochastic matrix is used to describe the transitions of a Markov chain.
- The stochastic matrix applied to a probability distribution redistributes the probability mass of the original distribution while preserving its total mass.
  - If this process is applied repeatedly, the distribution converges to a stationary distribution for the Markov chain.

  - A **stationary probability vector** I is defined as a vector that does not change under application of the transition matrix P.
- A right stochastic matrix is a square matrix of non-negative real numbers with each row summing to 1.
- One eigenvector of the transition matrix P has an eigenvalue of 1 and is all positive.
- A webgraph can be interpreted as a stochastic transition probability matrix in a markov chain.

Computing Transition Matrix P



$$A = \begin{pmatrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 \end{pmatrix}$$

$$D = \begin{pmatrix} 2 & 0 & 0 & 0 \\ 0 & 3 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 2 \end{pmatrix} \qquad D^{-1} = \begin{pmatrix} \frac{1}{2} & 0 & 0 & 0 \\ 0 & \frac{1}{3} & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & \frac{1}{2} \end{pmatrix}$$

$$\textit{Transition Matrix } Q = D^{-1}A = \begin{pmatrix} \frac{1}{2} & 0 & 0 & 0 \\ 0 & \frac{1}{3} & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & \frac{1}{2} \end{pmatrix} * \begin{pmatrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 \end{pmatrix} = \begin{pmatrix} 0 & \frac{1}{2} & 0 & \frac{1}{2} \\ \frac{1}{3} & 0 & \frac{1}{3} & \frac{1}{3} \\ 1 & 0 & 0 & 0 \\ \frac{1}{2} & 0 & \frac{1}{2} & 0 \end{pmatrix}$$

Figure 13: Transition matrix example

- Given a graph represented as an adjacency matrix A and an out degree matrix D, the transition matrix is computed as $P = D^{-1}A$
- Formallly,

$$\text{Adjacency Matrix, } A_{i,j} = \begin{cases} 1 & \text{if } (i,j) \in E \\ 0 & \text{Otherwise} \end{cases}$$

$$\text{Out-Degree Matrix, } D_{i,j} = \begin{cases} deg(i) & \text{if } i = j \\ 0 & \text{Otherwise} \end{cases}$$

$$\text{Transition Matrix, } P_{i,j} = \begin{cases} \frac{1}{deg(i)} & \text{if } (i,j) \in E \\ 0 & \text{Otherwise} \end{cases}$$

Stationary Distribution

- The probability distribution at time t+1 is given by $x_{t+1} = x_t P^T$
- A **stationary distribution** is defined as a vector that does not change under application of the transition matrix P.

$$I_0 = I_0 P^T$$

- This vector, I_o is the **Page rank vector**
  - It is the left eigen vector of P
  - It is also the largest eigen value in magnitude, equal to 1.
  - It is the only eigenvector with all nonnegative elements, proven by the Perron-Frobenius theorem.

Computing steady state distribution

- Let $I = (I_1, ..., I_n)$ denote the steady state probability distribution of n states.
- Let the transition matrix be denoted by P.
- If the current state is described by I, the next state is distributed as $I.P^T$
- But since I is the steady state, $I.P^T = I$
- **Power Method**: Multiply x by increasing powers of P^T until the product looks stable.
  - Regardless of where we start, we eventually reach the steady state I.
  - Assume x is any distribution.Say $(1, 0, 0, ...0)$
    - One step: xP^T
    - Two steps: $x(P^T)^2$
    - Three steps: $x(P^T)^3$
  - Eventually, for large "k", $x(P^T)^k = I$
- Will the above always work? - Based on Perron-Frobenius Theorem

Irreducible

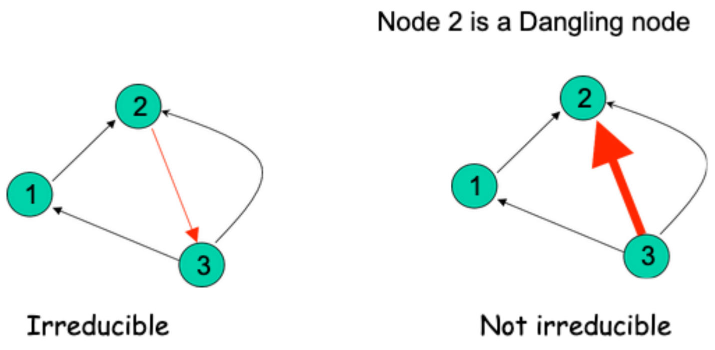


Figure 14: Irreducible example

- A graph is irreducible if there is a path from every node to every other node.
- A state i is **recurrent**, if starting at i and from where ever you go, there is a way of returning back to i.
- If a state is not recurrent, it is **transient**.
- Dangling nodes are transient.
- A collection of recurrent states that are connected to each other (and with no other state) is called a **recurrent class**.

Aperiodic

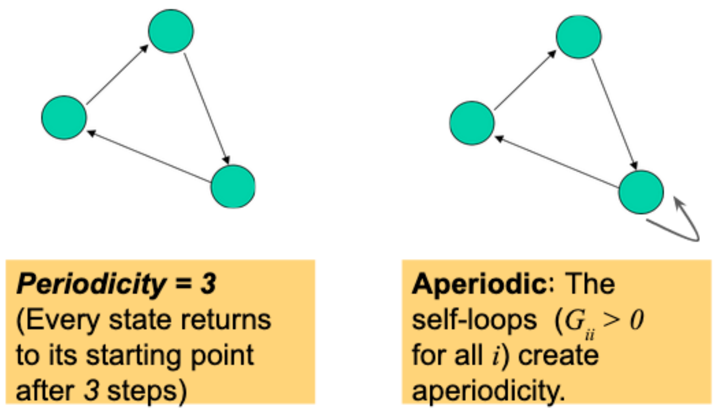

Periodicity = 3
(Every state returns to its starting point after *3* steps)
Aperiodic: The self-loops ($G_{ii} > 0$ for all i) create aperiodicity.

Figure 15: Periodicity example

- The GCD (greatest common ddivisor) of all cycle lengths is called the period.
- A node in a graph is periodic if starting at node i, we return to node i in “a” steps. Then periodicity of the node i is “a”.
- If the GCD is 1, then it is aperiodic.This can be achieved by adding a self-loop as shown above.
- A markov chain is aperiodic if every state is aperiodic.

- An irreducible Markov chain only needs one aperiodic state to imply all states are aperiodic.

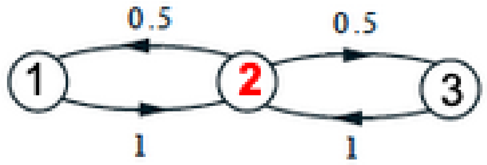


Figure 16: Periodicity example

- Consider the above example
- Starting at 2, there is a 0.5 probability of going either left or right.
- But, no matter left or right, you are back at 2 in the next step.
  - So every even step, we are back at 2. So probability is 1.
  - For every odd step, there is no way you are at 2, so probability is 0.
- After n iterations, ($n \to \infty$), the probability of state 2 doesn't converge to any fixed value.
  - It is still either 0 or 1 and keeps alternating between these values.

Steady state convergence theorem

If we start at 2 different states in a graph and randomly transition to different states, as long as we have a single recurrent class and no periodicity, the 2 paths will eventually collide. After this point, probabilistically, the future transitions of both paths are the same and the initial conditions stop having any influence.This is the steady state of the states.

Perron-Frobenius Theorem

- If a Markov Chain is irreducible and aperiodic, then the largest eigen value of the stochastic transition matrix will be equal to 1 and all other eigen values will be strictly less than 1.
- Formally, let the eigen values of P be $\sigma_i, i \in [0, ..., n-1]$ in non-increasing order of σ_i
- This gives us

$$\sigma_0 = 1 > |\sigma_1| \geq |\sigma_2| ... \geq |\sigma_{n-1}|$$

- The eigen vector, I_0 correspondding to σ_0 is invariant under the markov process. Thus,

$$I_0.P^T = I_0$$

Iterative Power Method

- If a markov chain is irreducible and aperiodic, the steady state probability vector I_0 can be found by the iterative power method.
- For any probability state x:
$$\lim_{n\to\infty} x.(P^T)^n = I_0$$

Implications of Perron-Frobenius Theorem:

- If a markov chain is irreducible and aperiodic then there exists a vector that is invariant,i.e. I_0
- If a markov chain is time homogenous, then the transition matrix, P is the same after each time-step.
  - So the k-step probability can be computed as the kth power of the transition matrix, i.e. P^k
- These imply, for a well-behaved graph (irreducible and aperiodic), there exists a unique stationary distribution.
- If we have the primitive transition matrix, we can use the power method to compute I_0

Using Page Rank

- Page rank is a query independent ranking of webpages.
- Once all webpages are ranked (with anumber between 0 and 1), we process queries from users as follows:
  - Retrieve all pages that match the input query.
  - Present the pages in order of their pre-computed page rank.

Computing Page Rank

- Markov processes give us a principled approach to calculate the PageRank of each web page;
- It is nothing more than the steady state probability distribution of the Markov process underlying the random-surfer model of web navigation.
- The PageRank is a link analysis algorithm that "measures" the relative importance of each page within the WebGraph.
- **Adjustments to make page rank work**
  - **Stochastic Adjustment**
    - In this adjustment, rows in the transition matrix for a dangling node (i.e a node with no neighbors. Have values of 0 in the row) are replaced by 1/N.
    - This makes the transition matrix stochastic.
  - **Primitivity Adjustment**
    - At dangling nodes, we use teleportation: This allows random transition to any page
  - So, we finally get: **States + Transition Matrix + Teleportion matrix = Markov Chain**
- **STEP 1**: Compute Stochastic transition matrix,P
  - Let Q be the transition matrix of the graph (with stochastic adjustment)
  - Let α be the probability of a random jump called **teleport probability**
  - Let N be the total number of nodes in the graph.
  - Let v be the biased teleport vector, computed as 1/N for every node. Called the **teleportation matrix**

$$P = \alpha . v + (1 - \alpha) Q$$

where, P is called the stochastic transition matrix

P is now irreducible and aperiodic

- **STEP 2**: Compute the page rank vector, π, such that,$\pi . P^T = \pi$
  - We use the power Iteration method to compute π
    - Initialize π to a random value or maybe 1/N
    - Compute as follows:

$$\pi^0 . P^T = \pi^1$$
$$\pi^1 . P^T = \pi^2$$

....

Until convergence

  - The final value of π is the page rank scores.

Distributed Page Rank

High level overview

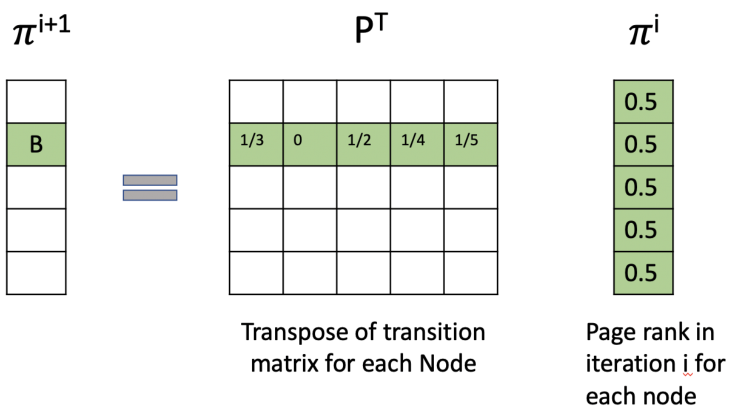
- In a distributed setting, operations will be centered around each node in the graph.
- We start by assigning a uniform probability to each node (1/N)
- In each iteration,
- MR JOB 1: Distribute page rank
  - Each node distributes its probability mass equally to all out-going links.
    - If A is assigned a probability of 0.5
    - If A has neighbors B,C and D
      - A assigns a probability mass of 0.5/3 to each of B, C and D.
    - If A is a dangling node, i.e it has no neighbors
      - It just emits a special string (like *) with all its probability mass
  - In the reducer,
    - for example, if key is B
      - The reducer sums all the probabilities assigned to B from various nodes.
      - This new value becomes the probability mass for node B
    - If the key found is *, it just sums up all the probability mass.
      - This is the total probability mass of all dangling nodes in the graph.
- MR JOB 2: Handle dangling nodes and teleportation
  - The mapper distributes the mass of dangling nodes to all the reducers
  - In the reducer, the dangling mass is distributed based on the teleportation or dampning factor α to each node.
  - The updated page rank for each node is then emiitted for the next step
- After multiple such iterations, the probabilities converge to a steady state value (within some pre-defined tolerance of change)
- This final vector of probabilities for each node will be the page rank distribution for the graph.

Mathematically,

- We need to find:

$$\pi^{i+1} = P^T \pi^i$$

- In each iteration, we are distributing the computation of the matrix multiplication as depicted below:



π^{i+1}
P^T
π^i
B
1/3
0
1/2
1/4
1/5
0.5
0.5
0.5
0.5
0.5
Transpose of transition matrix for each Node
Page rank in iteration i for each node

Figure 17: Page rank computation example

- In the MR Job 1,
  - Given a page X with inbound links $(t_1, t_2 ... t_n)$
  - Let C_t be the out degree of node t
  - PR(X) be the page rank of node X

$$PR(X) = \sum_{i=1}^{n} \frac{PR(t_i)}{C(t_i)}$$

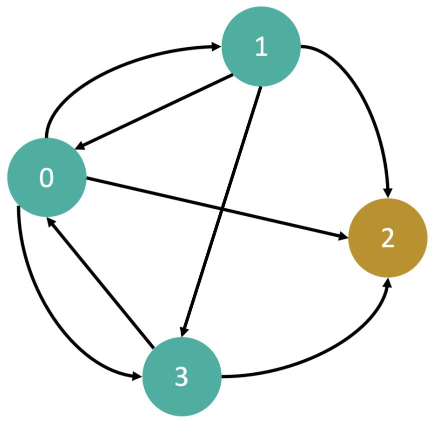
- From the distributed approach described, if X is B, t_1 would be A
  - $\frac{PR(A)}{C_A}$ is what each mapper emits as the prpobability mass of B
  - The reducer sums up all these values effectively computing PR(B)
- In MR Job 2,
  - Hyperparameter α is passed as input.N is the size of the graph.
  - For all the dangling nodes in the graph, the mapper would receive a probability mass of m.
  - The reducer will now compute the updated probability of a node as follows:

$$PR'(X) = \alpha . \frac{1}{N} + (1 - \alpha) . (\frac{m}{N} + PR(X))$$

  - In the single node computation of page rank, the above step is taken care while computing the stochastic transition matrix,P.
  - In the distributed version, this is done as an update to the computed page rank value as shown above.

Detailed Example

- Given the following graph:



$$\underset{D^{-1}}{\begin{bmatrix} 1/3 & 0 & 0 & 0 \\ 0 & 1/3 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1/2 \end{bmatrix}} \times \underset{A}{\begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 \end{bmatrix}} = \underset{Q}{\begin{bmatrix} 0 & 1/3 & 1/3 & 1/3 \\ 1/3 & 0 & 1/3 & 1/3 \\ 0 & 0 & 0 & 0 \\ 1/2 & 0 & 1/2 & 0 \end{bmatrix}}$$

Figure 18: Example Graph

- INIT: Create an adjacency list

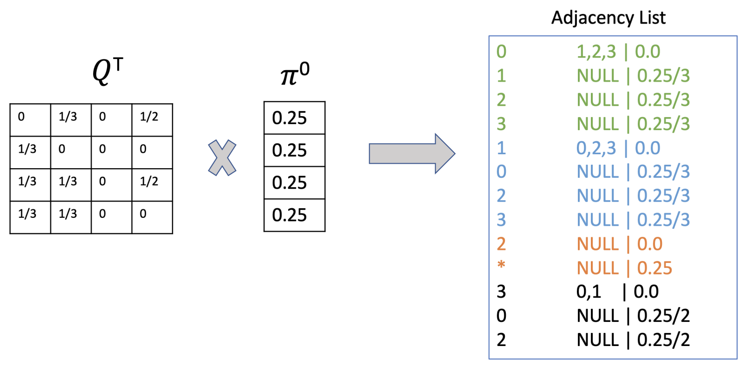
$$\underset{A}{\begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 \end{bmatrix}} \quad \underset{\pi^0}{\begin{bmatrix} 0.25 \\ 0.25 \\ 0.25 \\ 0.25 \end{bmatrix}} \Rightarrow$$

Adjacency List

0	1,2,3 | 0.25
1	0,2,3 | 0.25
2	NULL| 0.25
3	0,2 | 0.25

Figure 19: Input to MR Job 1

- MR JOB 1
  - The output from the mapper is as follows:



Q^T

0	1/3	0	1/2
1/3	0	0	0
1/3	1/3	0	1/2
1/3	1/3	0	0

π^0

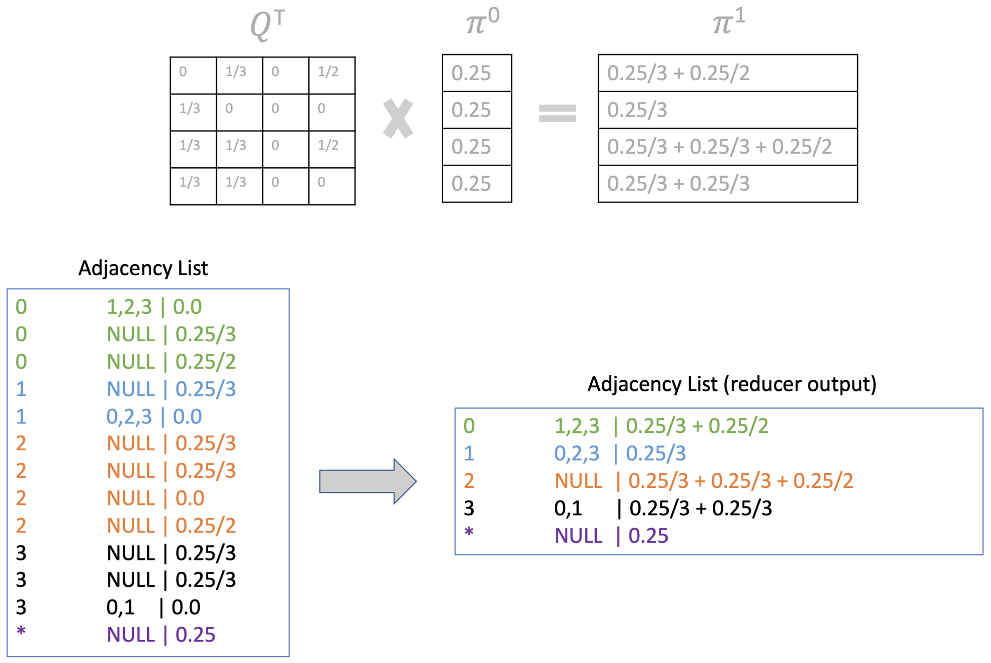
0.25
0.25
0.25
0.25

Adjacency List

```
0        1,2,3 | 0.0
1        NULL | 0.25/3
2        NULL | 0.25/3
3        NULL | 0.25/3
1        0,2,3 | 0.0
0        NULL | 0.25/3
2        NULL | 0.25/3
3        NULL | 0.25/3
2        NULL | 0.0
*        NULL | 0.25
3        0,1   | 0.0
0        NULL | 0.25/2
2        NULL | 0.25/2
```

Figure 20: MR1: Mapper output

- The output from the reducer is as follows:



Q^T

0	1/3	0	1/2
1/3	0	0	0
1/3	1/3	0	1/2
1/3	1/3	0	0

π^0

0.25
0.25
0.25
0.25

π^1

0.25/3 + 0.25/2
0.25/3
0.25/3 + 0.25/3 + 0.25/2
0.25/3 + 0.25/3

Adjacency List

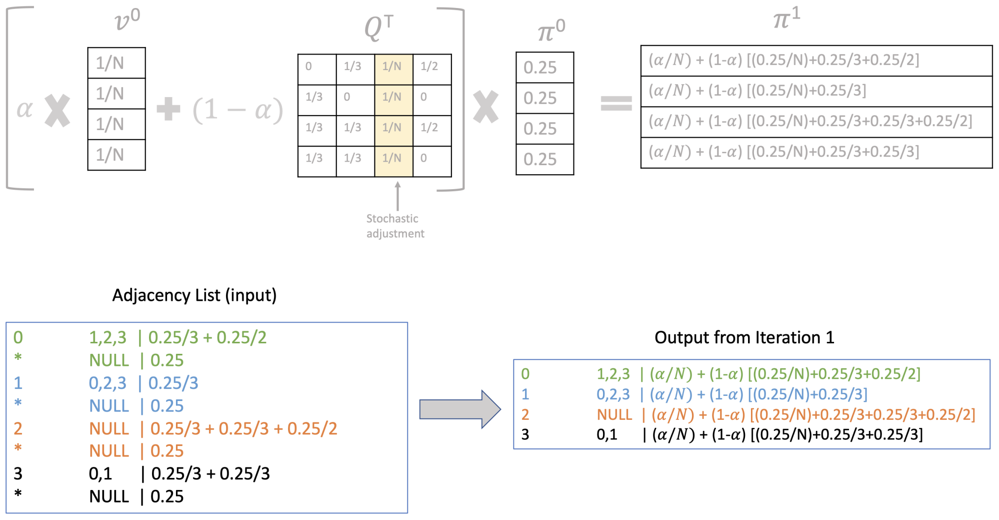
```
0        1,2,3 | 0.0
0        NULL | 0.25/3
0        NULL | 0.25/2
1        NULL | 0.25/3
1        0,2,3 | 0.0
2        NULL | 0.25/3
2        NULL | 0.25/3
2        NULL | 0.0
2        NULL | 0.25/2
3        NULL | 0.25/3
3        NULL | 0.25/3
3        0,1   | 0.0
*        NULL | 0.25
```

Adjacency List (reducer output)

```
0        1,2,3  | 0.25/3 + 0.25/2
1        0,2,3  | 0.25/3
2        NULL   | 0.25/3 + 0.25/3 + 0.25/2
3        0,1    | 0.25/3 + 0.25/3
*        NULL   | 0.25
```

Figure 21: MR1: Reducer output

- MR JOB 2
  - The mapper distributes the dangling node mass to each reducer.
  - The reducer computes as follows:



$$\left[\alpha \times \begin{bmatrix} 1/N \\ 1/N \\ 1/N \\ 1/N \end{bmatrix} + (1-\alpha) \begin{bmatrix} 0 & 1/3 & 1/N & 1/2 \\ 1/3 & 0 & 1/N & 0 \\ 1/3 & 1/3 & 1/N & 1/2 \\ 1/3 & 1/3 & 1/N & 0 \end{bmatrix}\right] \times \begin{bmatrix} 0.25 \\ 0.25 \\ 0.25 \\ 0.25 \end{bmatrix} = \begin{bmatrix} (\alpha/N) + (1\text{-}\alpha)\,[(0.25/N)+0.25/3+0.25/2] \\ (\alpha/N) + (1\text{-}\alpha)\,[(0.25/N)+0.25/3] \\ (\alpha/N) + (1\text{-}\alpha)\,[(0.25/N)+0.25/3+0.25/3+0.25/2] \\ (\alpha/N) + (1\text{-}\alpha)\,[(0.25/N)+0.25/3+0.25/3] \end{bmatrix}$$

Adjacency List (input)

```
0     1,2,3  | 0.25/3 + 0.25/2
*     NULL   | 0.25
1     0,2,3  | 0.25/3
*     NULL   | 0.25
2     NULL   | 0.25/3 + 0.25/3 + 0.25/2
*     NULL   | 0.25
3     0,1    | 0.25/3 + 0.25/3
*     NULL   | 0.25
```

Output from Iteration 1

```
0     1,2,3  | (α/N) + (1-α) [(0.25/N)+0.25/3+0.25/2]
1     0,2,3  | (α/N) + (1-α) [(0.25/N)+0.25/3]
2     NULL   | (α/N) + (1-α) [(0.25/N)+0.25/3+0.25/3+0.25/2]
3     0,1    | (α/N) + (1-α) [(0.25/N)+0.25/3+0.25/3]
```

Figure 22: MR2: Reducer output

- The two MR Jobs are run iteratively until convergence is reached.