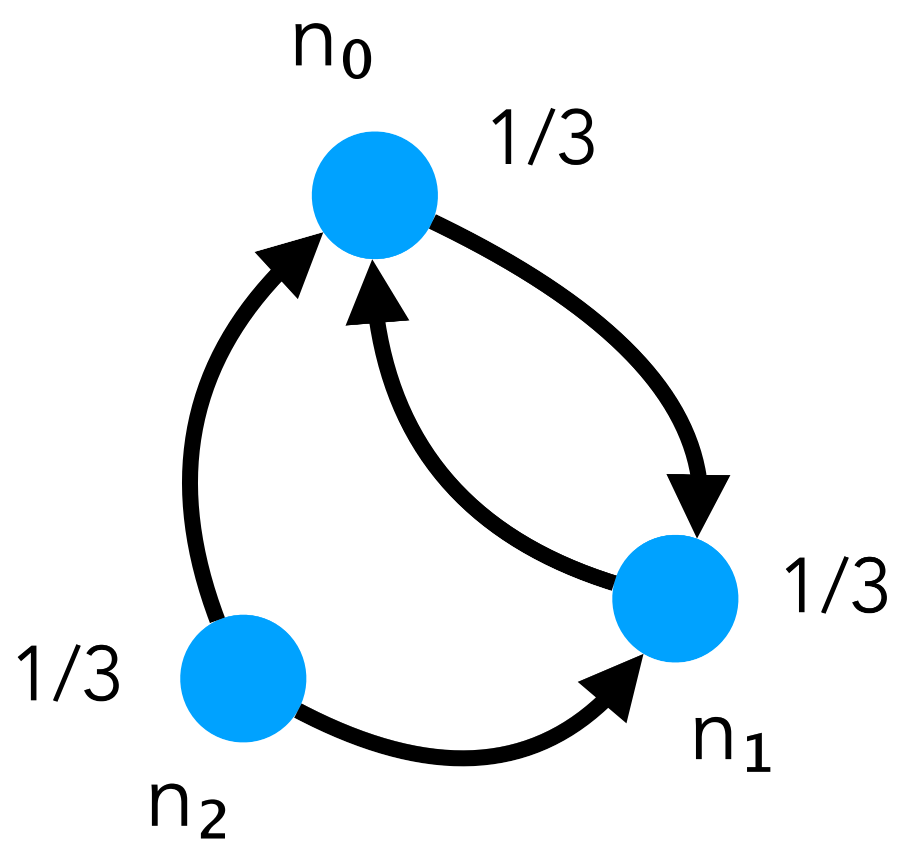
# The PageRank Algorithm

CIS 352 — Spring 2020

Kris Micinski

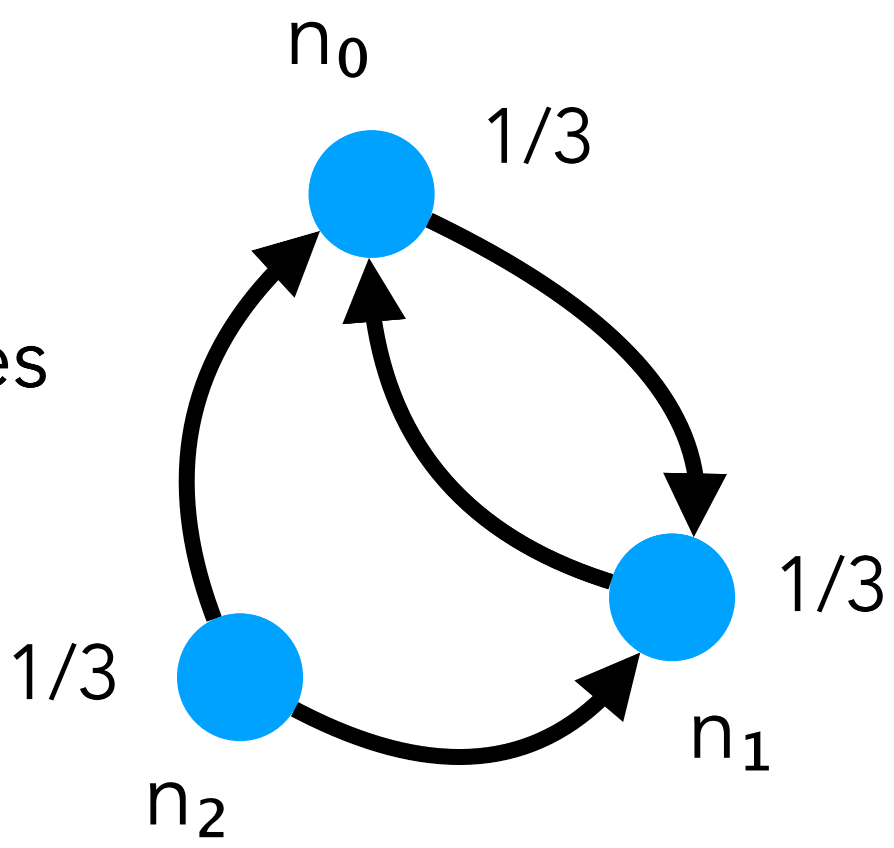
# Graphs



- A graph is a pair ⟨N,E⟩ of
  - A set of **nodes**, N
  - A set of **edges**, E, of the form
    - $(n_0,n_1) \mid n_0,n_1 \in N$
- Can equivalent represent in several ways:
  - Adjacency list (list of edges)
  - Graphs can be be composed of either **undirected** or **directed** edges

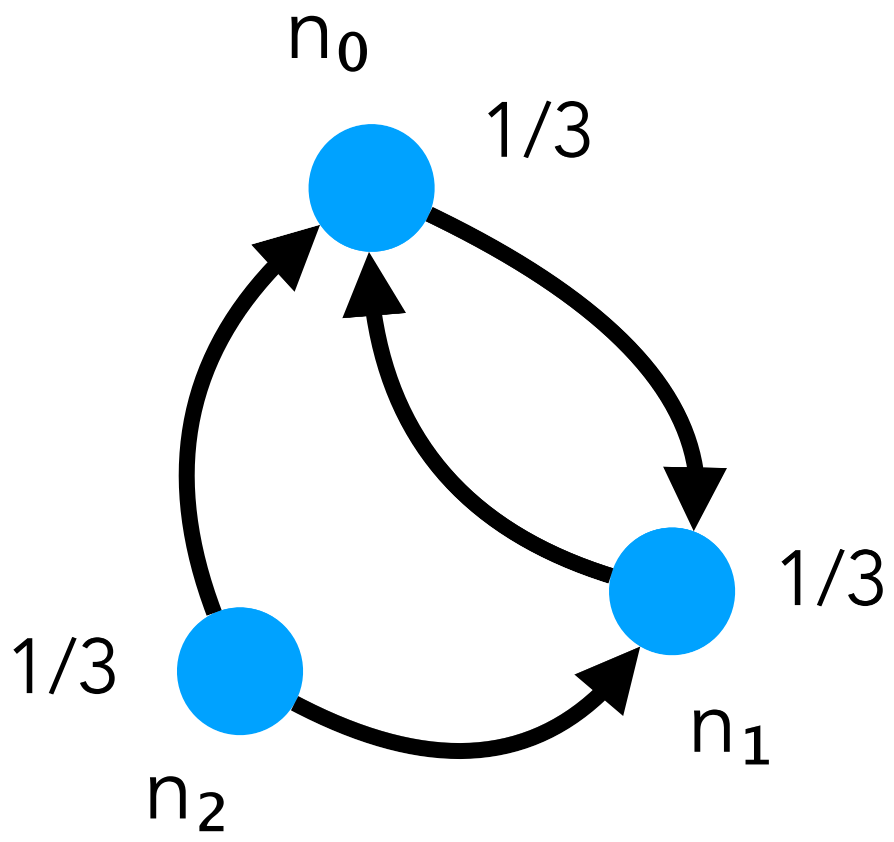
# PageRank

- Algorithm that originally powered Google
- Calculates a probability distribution on a graph
  - I.e., assigns a number in [0,1] to each node
  - This number is the page's "rank."
- Forms a **probability distribution**
  - Page ranks sum to 1 across all pages
  - $f \in N \rightarrow [0,1]$
  - $\sum_i f(i) = 1$ over $i \in \mathrm{dom}(f)$



# PageRank

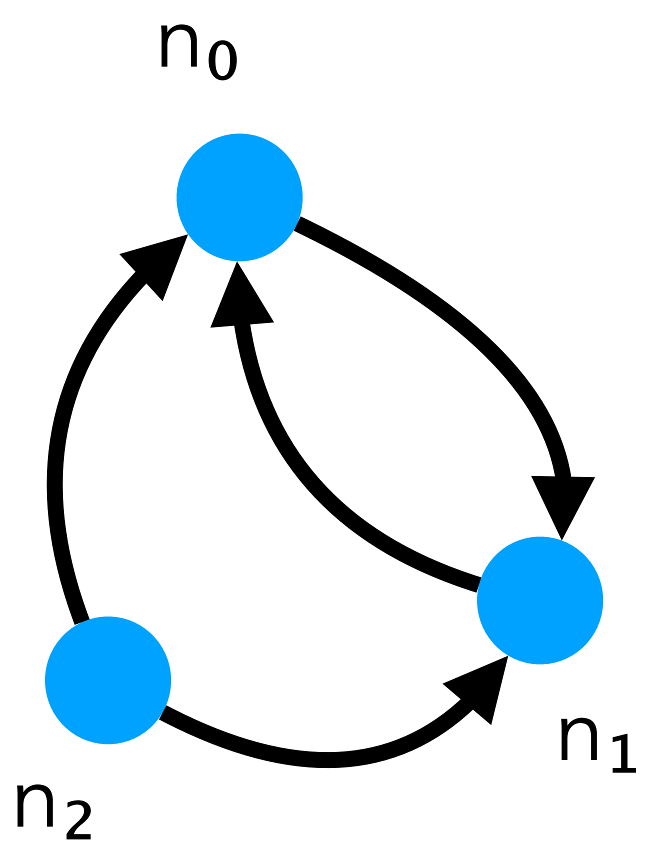
- Assumes a “random surfer” following links
- Calculates the likelihood of ending up at each page at any given point in time
- Popular pages have more incoming links
  - Thus have higher PageRank
- Iterative calculation over graph:
  - Each page gets “initial” PageRank
  - Then, iteratively update all PRs



# Representing Graphs in Racket

- For this assignment we will use **list of edges**
- Can use this to calculate:
  - Neighbors
  - Num of nodes in graph (total)
  - As input to PageRank

```
(define x '((n0 n1)
            (n1 n0)
            (n2 n0)
            (n2 n1)))
```



## Exercise

Write a function that calculates the pages to which a given page links

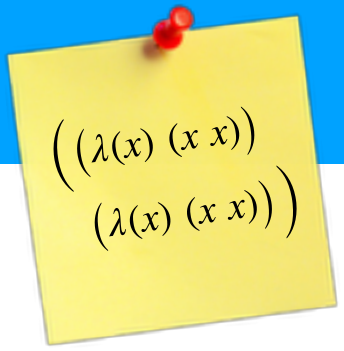
```
(define x '((n0 n1)
            (n1 n0)
            (n2 n0)
            (n2 n1)))
```

## Exercise

Write a function that calculates the pages to which a given page links

```
(define (links-of graph node)
  (define (loop graph l)
    (match graph
      [`() l]
      [`((,p0 ,p1) . ,rst)
       (if (equal? p0 node)
           (loop rst (cons p1 l))
           (loop rst l))]))
  (loop graph '()))
```

## Example



We might want to return a **set** of nodes, rather than a list

Sets are unordered collections

```
(define (links-of graph node)
  (define (loop graph l)
    (match graph
      [`() (set->list l)]
      [`((,p0 ,p1) . ,rst)
       (if (equal? p0 node)
           (loop rst (cons p1 l))
           (loop rst l))]))
  (loop graph '()))
```

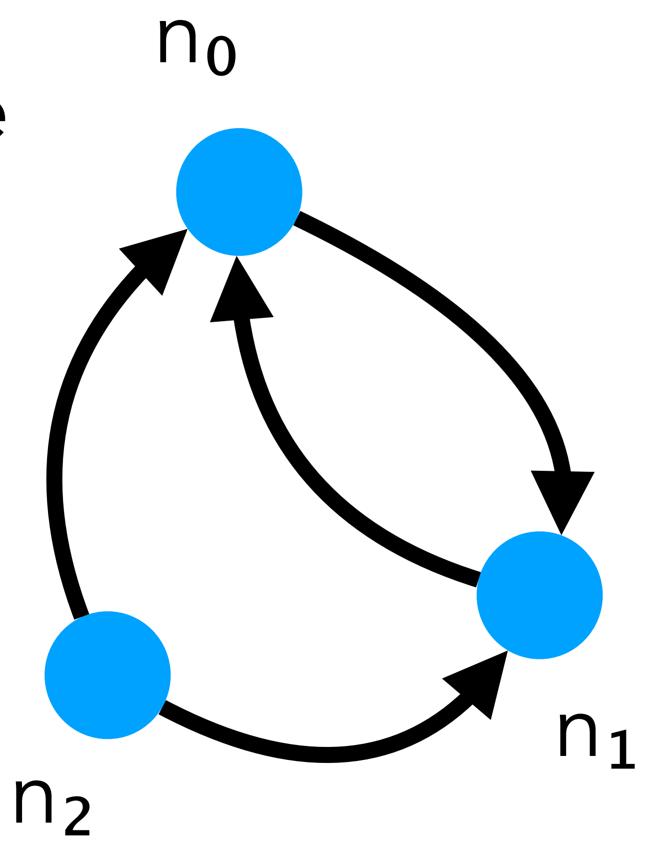
# Representing PageRanks

- PageRanks are represented using Racket **hashes**
  - Key/value maps (similar to hash tables)
  - Immutable w/ O(1) runtime for lookup/insert
    - Based on Hash Array-Mapped Tries (HAMT)
- `(hash ‘a 0 1 2 “hello” ‘c)` — creates hashes, note keys can be heterogeneous type
- `(hash-ref x ‘a)` — Looks up value for key `‘a`
- `(hash-set x ‘a 2)` — Returns a **new** hash with updated key for ‘a
- `(hash-keys x)` and `(hash-values x)`— Return list of keys / values (useful for iterating)

# PageRank algorithm

- Begins by constructing **initial** PageRank
  - Each page has rank 1/N (for N nodes)
- Then, performs an **iteration** step some number of times
  - You decide how long you want to do this
  - Usually until change is smaller than some delta

```
(hash ‘n0 1/3
      ‘n1 1/3
      ‘n3 1/3)
```



# PageRank Iteration Step

PageRank is like a **vote**. Each page has a certain share of votes (its PR), each step it votes for each page to which it links, but it divides its vote equally across links.

Intuitively the next PageRank for page i is the sum of:

- A **random chance** that a surfer will jump to i
  - (1-d)/N applies random chance to all pages
- The PageRanks of the pages that link to it, weighted by the number of links those pages have

$$PR(p_i) = \frac{1-d}{N} + d \sum_{p_j \in M(p_i)} \frac{PR(p_j)}{L(p_j)}$$

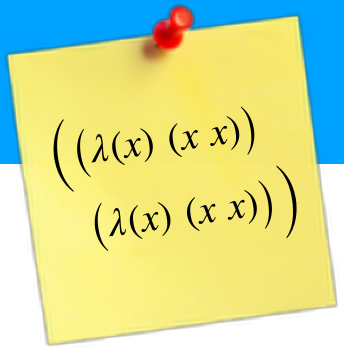
# PageRank Iteration Step

- At each step, the next PR for page i is calculated as:

$$PR(p_i) = \frac{1-d}{N} + d \sum_{p_j \in M(p_i)} \frac{PR(p_j)}{L(p_j)}$$

- Where:
  - $M(p_i)$: set of pages that link to i
  - $PR(p_j)$ and $PR(p_i)$: PageRanks of i and j
  - $L(p_j)$ is the number of links from j to any other page
  - d is a “dampening factor” (typically .85)

## Example

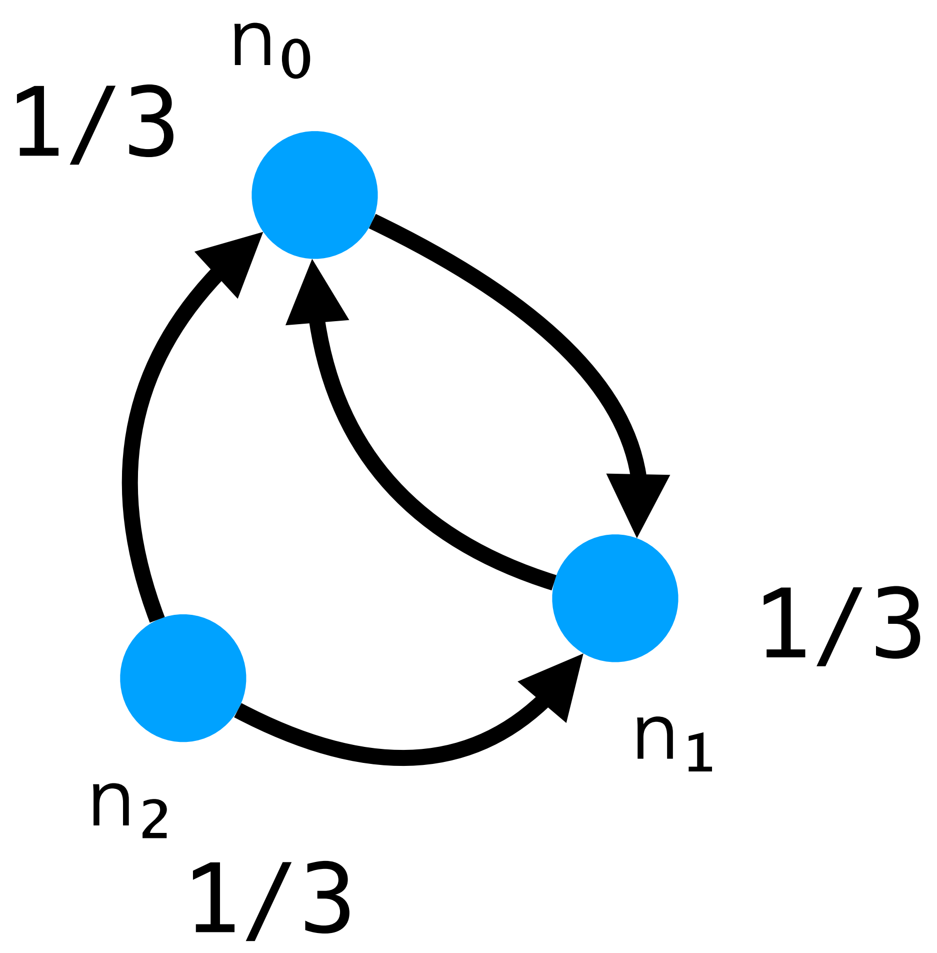


$$PR(p_i) = \frac{1-d}{N} + d \sum_{p_j \in M(p_i)} \frac{PR(p_j)}{L(p_j)}$$

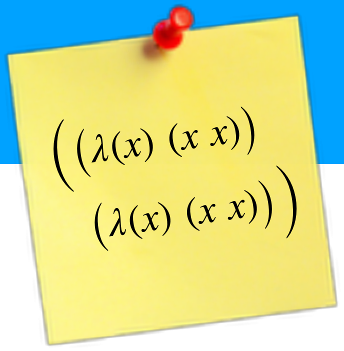
Let's calculate the next values of $n_0$, $n_1$, and $n_2$ (assume d=85/100)

For $n_0$. Sum of…

- (1-85/100)/3, since 3 nodes
- For $n_2$…
  - 85/100 * 1/3 / 2
- For $n_1$…
  - 85/100 * 1/3 / 1
- = 19/40



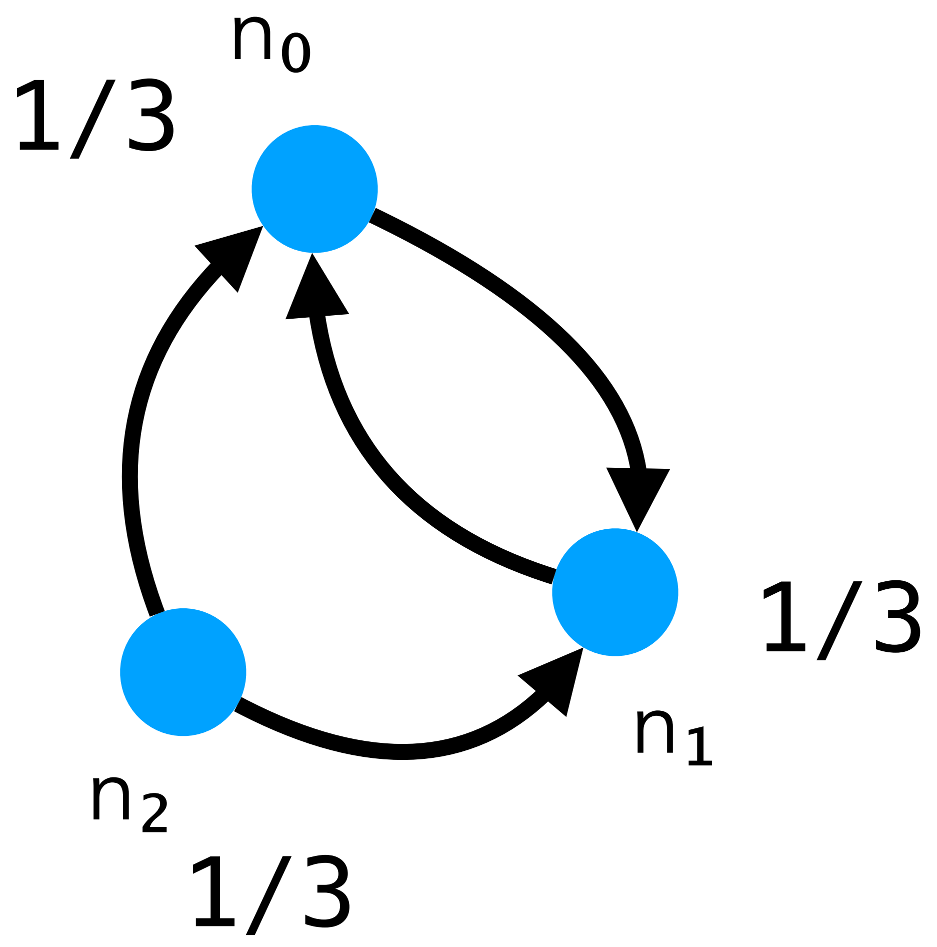
## Example



$$PR(p_i) = \frac{1-d}{N} + d \sum_{p_j \in M(p_i)} \frac{PR(p_j)}{L(p_j)}$$

So next PR should be…

```
(hash 'n0 19/40
      'n1 19/40
      'n2 1/20)
```



# PageRank Assumptions

- Several simplifying assumptions for input graphs
- No “self-links:” remove links from a page to itself
- All nodes link to at least one other node
  - Can fix this manually: link to **every other node**
- These steps necessary to make math work out (i.e., so that iteration forms a probability distribution)
- **All test input graphs have this form**

# Hints

- Read Racket docs for lists, sets, and hashes
- **Start sooner rather than later**
  - Will require much more time than a0
- num-pages, num-links, and num-backlinks are all easier
  - Should be able to mostly do now
- mk-initial-pagerank, step-pagerank and itaerate-pagerank-until are a little harder