

ENGR 102

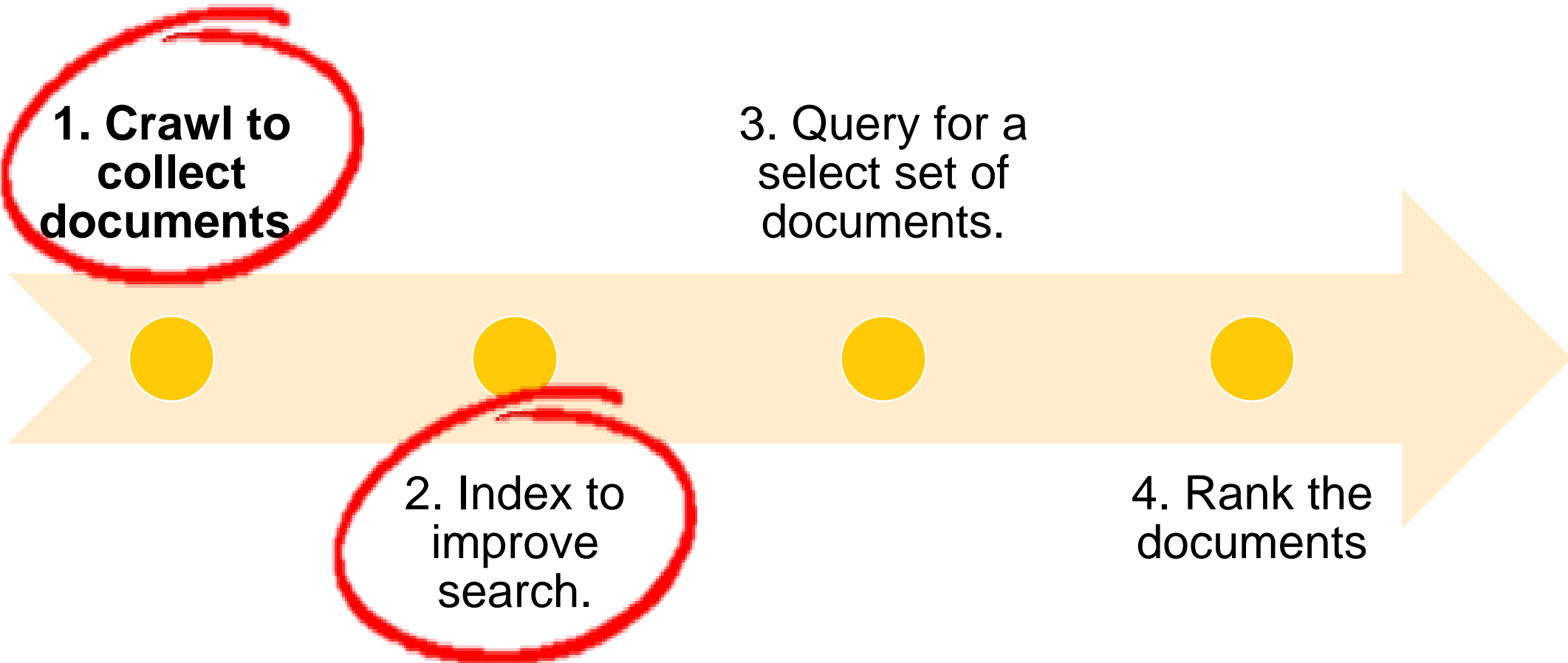
PROGRAMMING

PRACTICE

WEEK 12

Searching & Ranking

Search Engine



Search Engine

- Create a Python module (`mysearchengine.py`).
- The module will have two classes:
 - one for crawling and creating the database, and
 - the other for doing full-text searches by querying the database, as well as ranking.

Crawling pages - crawl()

```
def crawl(self, pages, depth=2):  
    for i in range(depth):  
        newpages = set()  
        for page in pages:  
            r = requests.get(page)  
            soup = BeautifulSoup(r.content)  
            if not self.addtoindex(page, soup):  
                continue  
  
            links = soup.find_all('a')  
            for link in links:  
                if 'href' in link.attrs:  
                    url = urljoin(page, link['href'])  
  
                    if not self.isindexed(url):  
                        newpages.add(url)  
  
                    linkText = self.gettextonly(link)  
                    self.addlinkref(page, url, linkText)  
  
        pages = newpages
```

End loop

Recording links

Add a link between two pages

```
def addlinkref(self, urlFrom, urlTo, linkText):  
    fromUrl = smart_str(urlFrom)  
    toUrl = smart_str(urlTo)
```

```
if fromUrl == toUrl: return False
```

```
# {toUrl: {fromUrl: None}}
```

```
self.link.setdefault(toUrl, {})  
self.link[toUrl][fromUrl] = None
```

```
words = self.separatewords(linkText)
```

```
for word in words:  
    word = smart_str(word)
```

```
if word in ignorewords: continue
```

```
# {word: [(urlFrom, urlTo), (urlFrom, urlTo), ..., (urlFrom, urlTo)]}  
self.linkwords.setdefault(word, [])  
self.linkwords[word].append((fromUrl, toUrl))
```

```
return True
```

Crawling pages

```
import mysearchengine
```

```
pagelist=['http://sehir.edu.tr/en']
```

```
dbtables = { 'urllist': 'urllist.db',  
             'wordlocation': 'wordlocation.db',  
             'link': 'link.db', 'linkwords': 'linkwords.db',  
             'pagerank': 'pagerank.db' }
```

```
crawler = mysearchengine.Crawler(dbtables)  
crawler.createindextables()
```

```
crawler.crawl(pagelist)
```

Search Engine

1. Crawl to collect documents.

2. Index to improve search.

3. Query for a select set of documents.

4. Rank the documents

Querying

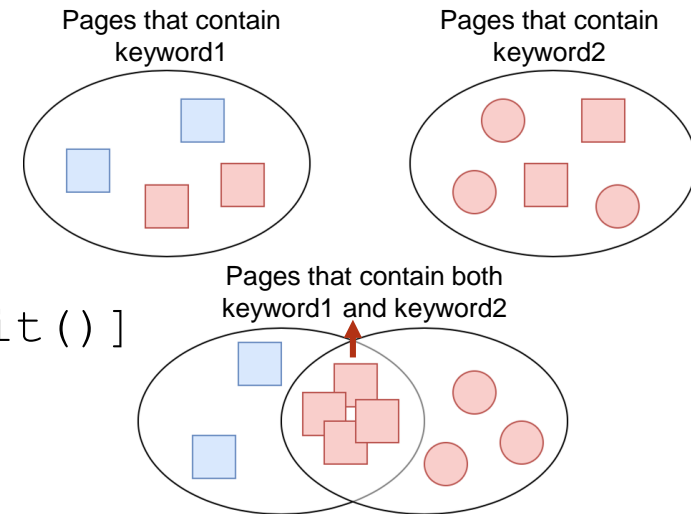
```
def getmatchingpages(self, q):
    results = {}
    # Split the words by spaces
    words = [smart_str(word) for word in q.split()]
    if words[0] not in self.wordlocation:
        return results, words

    #{word:{url:[loc1, loc2, ..., locN]}}
    url_set = set(self.wordlocation[words[0]].keys())

    for word in words[1:]:
        if word not in self.wordlocation:
            return results, words
        url_set = url_set.intersection(self.wordlocation[word].keys())

    for url in url_set:
        results[url] = []
        for word in words:
            results[url].append(self.wordlocation[word][url])

    return results, words
```



Querying

- We've managed to retrieve pages that match the queries.
- **Problem:** In a large set of pages, you would be stuck sifting through *a lot of irrelevant content* for any mention of each of the query terms in order to find the pages that are **really related to your search**.

Search Engine

1. Crawl to collect documents.

3. Query for a select set of documents.

2. Index to improve search.

4. Rank the documents

Ranking

Baseline – no ranking

```
import mysearchengine
```

```
pagelist=[ 'http://sehir.edu.tr/en' ]
```

```
dbtables = { 'urllist': 'urllist.db',  
             'wordlocation': 'wordlocation.db',  
             'link': 'link.db', 'linkwords': 'linkwords.db',  
             'pagerank': 'pagerank.db' }
```

```
searcher = mysearchengine.Searcher(dbtables)
```

```
searcher.query('engineering')
```

Ranking

- Content-based Ranking
- Inbound-link Ranking

Ranking measures

Content-based

- **Word frequency**: the number of times the words in the query appear in the document can help determine how relevant the document is.
- **Document location**: the main subject of a document will probably appear near the beginning of the document.
- **Word distance**: if there are multiple words in the query, they should appear close together in the document.

Ranking

word frequency - based

- The word frequency metric scores a page based on how many times the words in the query appear on that page.
- Search for “python”
 - a page about *Python* (or *pythons*) with many mentions of the word *python*.

vs.

- a page about a musician that has a pet python.

Ranking

word frequency - based

```
def frequencyscore(self, results):  
    counts = {}  
  
    # results: {url:[[loc1, loc2, ...], [locX, locY, ...], [...]]}  
  
    for url in results:  
        score = 1  
  
        for wordlocations in results[url]:  
            score *= len(wordlocations)  
  
        counts[url] = score  
  
    return self.normalizescores(counts, smallIsBetter=False)
```


Normalization

- In order to compare the results from different scoring methods, we need a way to normalize them.
- The normalization function:
 - each score is scaled according to how close it is to the best result, which will always have a score of 1.

```
# scores: {url: score}
```

```
normalizescores(scores, smallIsBetter=False)
```

False

$$\frac{\text{score}}{\max(\text{scores})}$$

True

$$\frac{\min(\text{scores})}{\text{score}}$$

Score Computation

```
def getscoredlist(self, results, words):  
  
    totalscores = dict([(url, 0) for url in results])  
  
    # word frequency scoring  
    weights = [(1.0, self.frequency_score(results))]  
    #           (1.0, self.location_score(results)),  
    #           (1.0, self.inboundlink_score(results))]  
  
    for (weight, scores) in weights:  
        for url in totalscores:  
            totalscores[url] += weight*scores.get(url, 0)  
  
    return totalscores
```

Ranking

word frequency - based

change the weights line in `getscoredlist` to this:

```
weights=[ (1.0, self.frequency_score(results)) ]
```

```
import mysearchengine
pagelist=[ 'http://sehir.edu.tr/en' ]

dbtables = { 'urllist': 'urllist.db',
              'wordlocation': 'wordlocation.db',
              'link': 'link.db', 'linkwords': 'linkwords.db',
              'pagerank': 'pagerank.db' }

searcher = mysearchengine.Searcher(dbtables)

searcher.query('engineering')
```

Ranking

word location - based

- Search term's location in the page.
- If a page is relevant to the search term, it will appear closer to the top of the page
 - perhaps even in the title.
- Score results higher if the query term appears early in the document.

```
# results: {url:[[loc1, loc2, ...], [locX, locY, ...], [...]]}
```

Ranking

word location - based

```
def locationscore(self, results):  
    locations = {}  
  
    # results: {url:[[loc1, loc2, ...], [locX, locY, ...], [...]]}  
  
    for url in results:  
        score = 0  
  
        for wordlocations in results[url]:  
            score += min(wordlocations)  
  
        locations[url] = score  
  
    return self.normalizescores(locations, smallIsBetter=True)
```

Ranking

word location - based

change the weights line in getscoredlist to this:

```
weights=[ (1.0, self.locationscore(results)) ]
```

```
import mysearchengine
pagelist=[ 'http://sehir.edu.tr/en' ]

dbtables = { 'urllist': 'urllist.db',
              'wordlocation': 'wordlocation.db',
              'link': 'link.db', 'linkwords': 'linkwords.db',
              'pagerank': 'pagerank.db' }

searcher = mysearchengine.Searcher(dbtables)

searcher.query( 'engineering' )
```

Ranking

word distance - based

- When a query contains multiple words, seek results in which the words in the query are close to each other in the page.
 - i.e., seek pages that conceptually relates search words.

Ranking

word distance - based

Left as a take-home exercise!

Link information

- Used scoring metrics based on the content of the page.
- How about information that others have provided about the page?
 - who linked to the page?
 - what they said about it?

Pages created by spammers are less likely to be linked than pages with real content.

Link information

link stores two URL IDs, indicating a link from one page to another.

```
{tourl: {fromUrl: None}}
```

linkwords store words that are included in a link.

```
{word: [(urlFrom1, urlTo1), ..., (urlFromN, urlToN)]}
```

- The **link** table has the URLs for the source and target of every link that it has encountered.
- The **linkwords** table connects the words with the links.

Simple Count

- Count inbound links to each page and use the total number of links as a metric for the page.
- The scoring function based on this count:

```
def inboundlinkscore(self, results):  
    inboundcount=dict([(url, len(self.link[url]))  
                        for url in results if url in self.link])  
  
    return self.normalizescores(inboundcount)
```

Simple Count

- Using this metric by itself will simply return all the pages containing the search terms,

“ranked solely on how many inbound links they have.”

- We need to combine the inbound-links metric with one of the content/relevance metrics we saw earlier.

```
weights = [ (1.0, self.frequency_score(results)),  
            (1.0, self.location_score(results)),  
            (1.0, self.inboundlink_score(results)) ]
```

What if?

- Someone can easily set up several sites pointing to a page whose score they want to increase.

What about ...?

- What about result pages that have attracted the attention of very **popular** sites?
- How to make links from popular pages worth more in calculating rankings?

PageRank

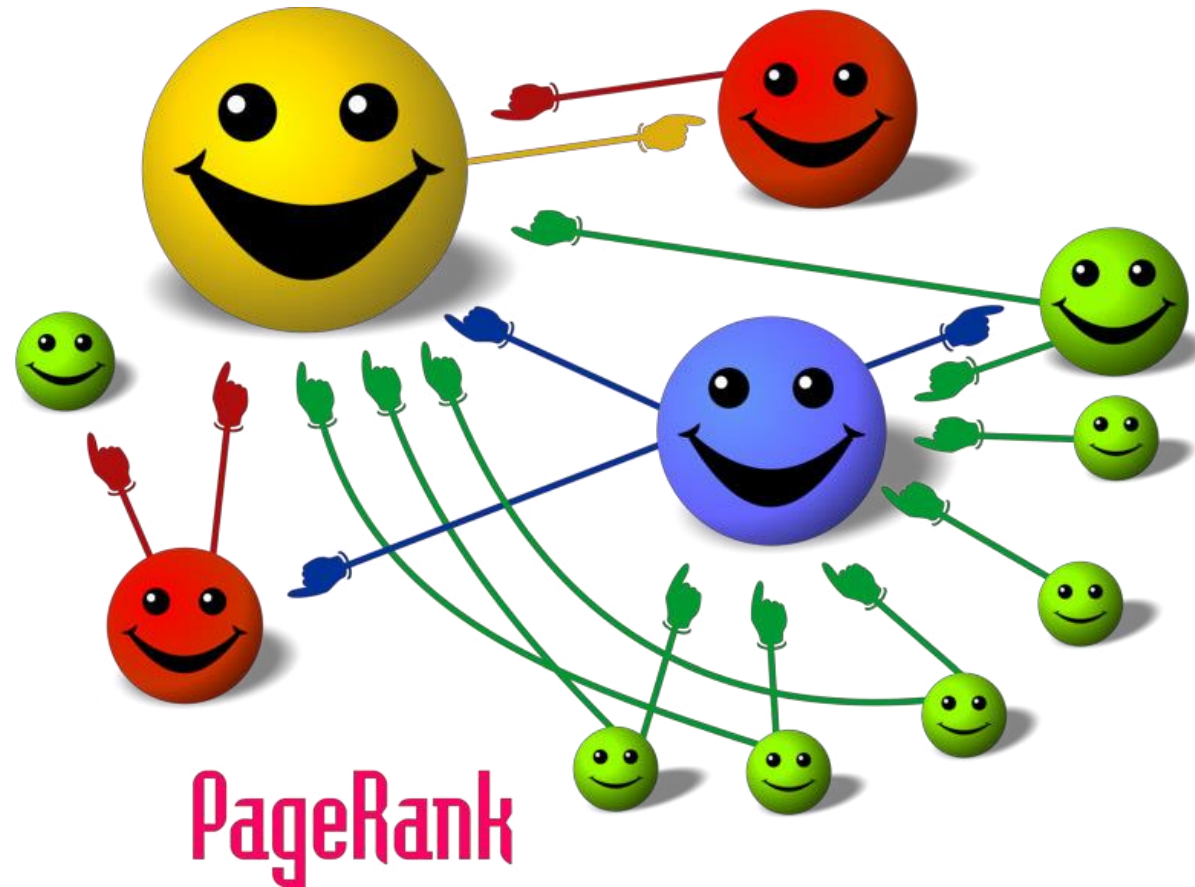
PageRank

- **PageRank** (named after Larry **Page**) calculates the probability that someone randomly clicking on links will arrive at a certain page.
- The more inbound links the page has from other popular pages, the more likely it is that someone will end up there purely by chance.

PageRank

- Basic principle of PageRank.

The size of each face is proportional to the total size of the other faces which are pointing to it.



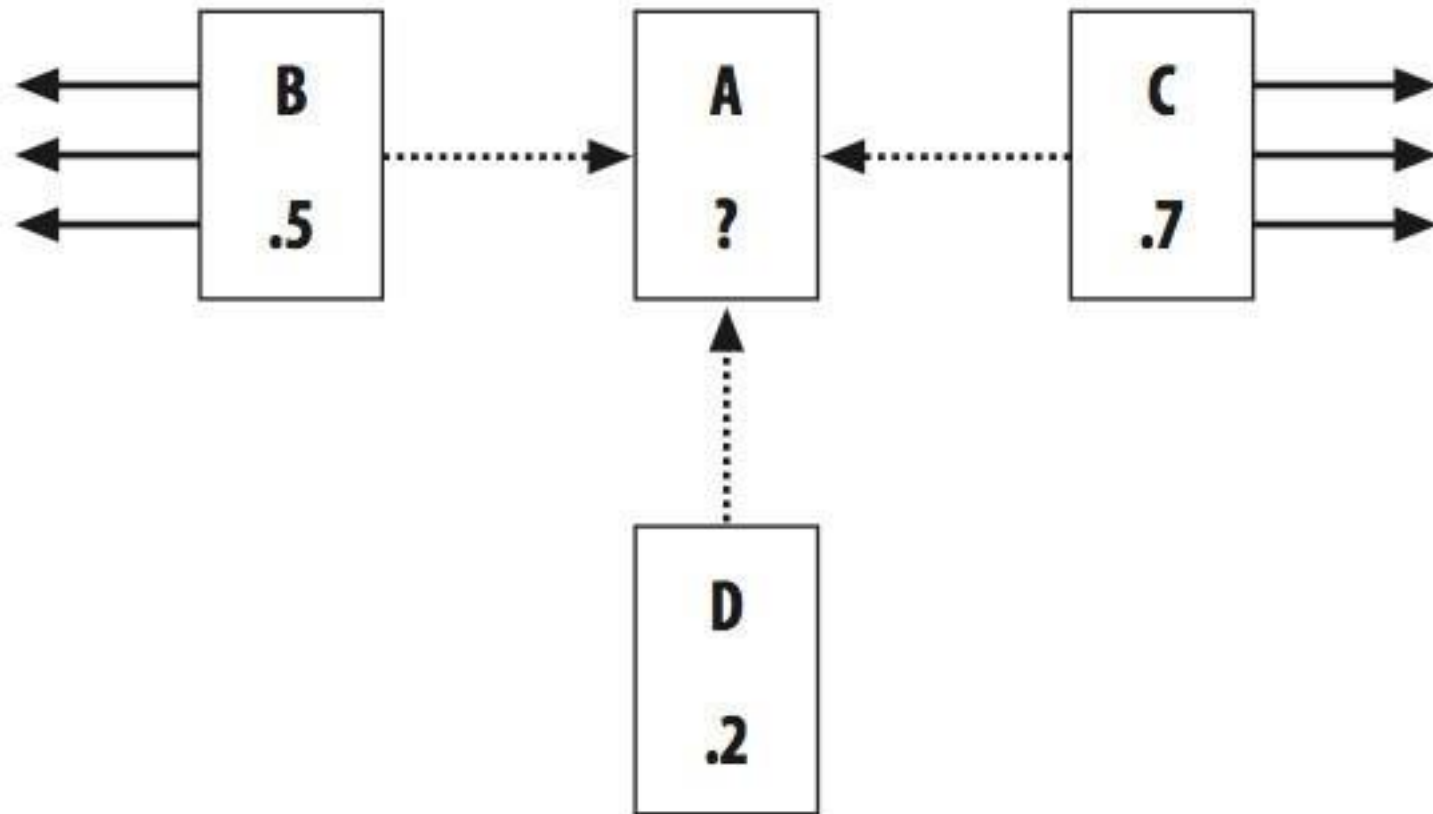
Source: <http://en.wikipedia.org/wiki/PageRank>

PageRank

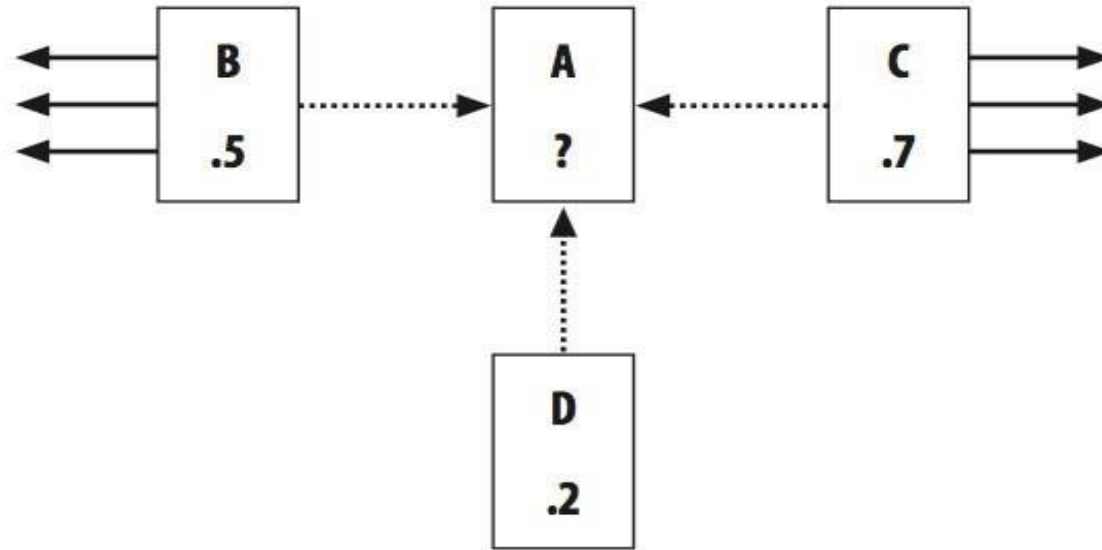
- If the user keeps clicking forever, they'll eventually reach every page, but most people stop surfing after a while.
- To capture this, PageRank also uses a damping factor of 0.85, indicating that

there is an 85% chance that a user will continue clicking on links at each page.

PageRank



PageRank



$$\text{PR}(A) = 0.15 + 0.85 * (\text{PR}(B)/\text{links}(B) + \text{PR}(C)/\text{links}(C) + \text{PR}(D)/\text{links}(D))$$

$$= 0.15 + 0.85 * (0.5/4 + 0.7/4 + 0.2/1)$$

$$= 0.15 + 0.85 * (0.125 + 0.175 + 0.2)$$

$$= 0.15 + 0.85 * 0.5$$

$$= 0.575$$

There is a small catch!

- All the pages linking to A already had PageRanks.
- You can't calculate a page's score until you know the scores of all the pages that link this page!
 - And, you can't calculate their scores without doing the same for all the pages that link to them.
- How is it possible to calculate PageRanks for a whole set of pages that don't already have PageRanks?



AND YET THE QUESTION REMAINED:
"WHO CAME FIRST?"

PageRank

- The solution is to set all the PageRanks to an initial arbitrary value
 - *(the code will use 1.0, but the actual value doesn't make any difference).*
- Repeat the calculation over several iterations.
- After each iteration, the PageRank for each page gets closer to its true PageRank value.
- The number of iterations needed varies with the number of pages.

PageRank computation

```
def calculatepagerank(self, iterations=20):
    self.pagerank = shelve.open(self.dbtables['pagerank'],
                                writeback=True, flag='n')

    # initialize every url with a page rank of 1
    for url in self.urllist:
        self.pagerank[url] = 1.0

    for i in range(iterations):
        print "Iteration %d" % (i)
        for url in self.urllist:
            pr = 0.15
            # Loop through all the pages that link to this one
            if url in self.link:
                for linker in self.link[url]:
                    linkingpr = self.pagerank[linker]

                    # Get the total number of links from the linker
                    linkingcount = self.urllist[linker]
                    pr += 0.85*(linkingpr/linkingcount)

            self.pagerank[url] = pr
```


PageRank computation

```
import mysearchengine
```

```
pagelist=['http://sehir.edu.tr/en']
```

```
dbtables = {'urllist': 'urllist.db',  
            'wordlocation': 'wordlocation.db',  
            'link': 'link.db', 'linkwords': 'linkwords.db',  
            'pagerank': 'pagerank.db' }
```

```
crawler = mysearchengine.Crawler(dbtables)  
crawler.createindextables()
```

```
crawler.calculatepagerank()
```

Score Combination

```
weights = [ (1.0, self.locationscore(rows)),  
            (1.0, self.frequency_score(rows)),  
            (1.0, self.pagerank_score(rows)) ]
```