Search Engines

Dr. Johan Hagelbäck

johan.hagelback@lnu.se

http://aiguy.org

Search Engines

- This lecture is about full-text search engines, like Google and Microsoft Bing
- They allow people to search a large number of documents for a word or set of words, and rank the results based on how relevant documents are to the search query
- Improving search engines is a very important area in computer science, and companies like Google, Microsoft and Yahoo have spent huge amounts of money and resources in it
- Google started as an academic project describing the PageRank algorithm for improving result ranking

Search Engines

- Searching and Ranking is a huge area that has been around for a very long time, and we can only cover some key concepts here
- We will only use a relatively small data set consisting of the Wikipedia articles
  - 250 articles about video games and 400 about programming
- Search engines for huge amounts of data like the Internet have many additional problems (for example storage) that we will not cover here

The data set

- A search engine needs a set of documents that the user can search in
- Search engines generate this set by using a *web crawler*
- The set can also be a collection of business documents in a company, which employees can search in

Indexing a document

- Once we have retrieved the contents of a document it must be *indexed*
- This involves creating a list of all words that appear in the document, and the location of each word
- We can then merge the word lists for all documents in a large table
- In case of HTML files, we must strip the file contents from all code and create a bag-of-words representation

Indexing

- The beginning of the index for the *Action_game* Wikipedia page will then look like:

  action game part of a series onaction games subgenres actionadventure game action roleplaying game open world stealth game survival game ...

- This representation has quite high storage requirements
- An optimization is to give each word a unique value, for example using hash codes, and store a list of integers instead
- In this case we must also store a table linking words to hash values

Searching

- Searching for all documents that match a search query is straightforward once we have an index
- If we search for a single word, the system returns a list containing all documents that has the word in them
- If we search for multiple words, the system returns a list of documents that has any of the words in them
  - Or perhaps only documents that have all words in them
  - It depends on the purpose of the search engine. How inclusive/exclusive shall the system be?

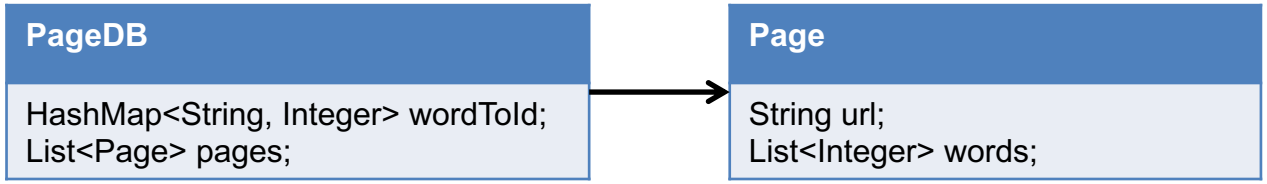
Ranking

- The interesting and complex problem is how we can sort the result list of matching documents based on how relevant each document is to the search query
- To do this we need to generate a numeric score for each document's relevance
- This score is typically a combination of several metrics
- There are lots of different metrics that can be used, and we will take a look at some of them in this lecture
- A metric can also typically be tweaked to hopefully work better for a specific search engine

Search Engine basics

- The first step is to build the basic structure of our search engine
- First we need to generate the index for all Wikipedia pages
- We can store this in for example a SQL database or, for smaller sets, in the main memory
- In our example we will generate the index when starting the system and store it in the main memory
- The word lists will be stored as integer values, so we need a mapping from word to values

Search Engine structure



From word to id

- We will have an integer counter that is increased by 1 every time we have to add a unique word
- Instead of storing this counter, we can use the size of the hash map as counter:

```
//Get Id for a word
int getIdForWord(String word)
  if (wordToId.containsKey(word))
    //Word found in hashmap
    return wordToId.get(word)
  else
    //Add missing word to hashmap
    int id = wordToId.size()
    wordToId.put(word, id)
    return id
```


Generate the word list

- To generate the word list we read the bag-of-words data files in the Wikipedia data set zip file
- Create one Page object for each file in the *data/Words/Games* and *data/Words/Programming* folders
- The Page object shall contain:
  - The URL to the page
  - The words in the page

Searching

- When searching for a query, we convert the search query to a sequence of word id values using the *getIdForWord(String)* method
- We then return all documents that has any of the word id values in their words *list*

- Now we have a search result, next step is to sort it!

Order by content

Content-Based Ranking

- The first approach we will use is *Content-Based* ranking
- It is metrics that sort the search results based on the contents of each page
- We will look at three different metrics:
  - Word frequency
  - Document location
  - Word distance
- Earlier search engines often only used Content-Based ranking, and could give useable results

Basic query structure

- The basic query structure code shall work like this:
  - Create an empty list where we place the results
  - For each page:
    - Calculate the metric value for each of the metrics used
  - Normalize the metric values so all metrics are between 0 and 1
  - Calculate a final score that is a weighted sum of all normalized metric values

- The code can look something like this:

Query algorithm

```
void query(String query)
  result = new List()
  Score scores

  //Calculate score for each page in the pages database
  for (i = 0 to pagedb.noPages())
    Page p = pagedb.get(i)
    scores.content[i] = getFrequencyScore(p, query)
    scores.location[i] = getLocationScore(p, query)

  //Normalize scores
  normalize(scores.content, false)
  normalize(scores.location, true)

  //Generate result list
  for (i = 0 to pagedb.noPages())
    Page p = pagedb.get(i)
    //Only include results where the word appears at least once
    for (scores.content[i] > 0)
      //Calculate sum of weighted scores
      double score = 1.0 * scores.content[i] + 0.5 *
scores.location[i]
      result.add(Score(p, score))

  //Sort result list with highest score first
  sort(result)

  //Return result list
  return result
```

AND / OR

- The query algorithm include all pages where any search query word appears at least once on the page
- This is equivalent to:

  super **OR** mario

- Another option is to include all pages where all search query words must appear at least once on the page
- This is equivalent to:

  super **AND** mario

Normalization

- We need all metric values to have a score between 0 and 1, where 0 is really bad and 1 is really good
- The problem is that some metric functions give low values for good matches, and others high values for good matches
- We need a universal function that converts the metric value to a score between 0 and 1, regardless of if high values are good or bad
- The code for doing this can look like this:

Normalization algorithm

```
void normalize (double[] scores, bool smallIsBetter)
  if (smallIsBetter)
    //Smaller values shall be inverted to higher values
    //and scaled between 0 and 1
    //Find min value in the array
    double min_val = Min(scores)
    //Divide the min value by the score
    //(and avoid division by zero)
    for (i = 0 to scores.length())
      scores[i] = min_val / Max(scores[i], 0.00001)
  else
    //Higher values shall be scaled between 0 and 1
    //Find max value in the array
    double max_val = Max(scores)
    //To avoid division by zero
    max_val = Max(max_val, 0.00001)
    //When we have a max value, divide all scores by it
    for (i = 0 to scores.length())
      scores[i] = scores[i] / max
```


Word Frequency metric

- This metric scores a page based on how many times each word in the search query appears on the page:

```
void word_freq(Page p, String query)
  //Split search query to get each word
  String[] qws = query.split(" ")
  double score = 0
  //Iterate over each word in the search query
  foreach (String q : qws)
    //Iterate over all words in the page
    foreach (String word : p.words())
      //Increase score by one if the page word matches
      //the query word
      if (word == q)
        score += 1
  //Return the score
  return score
```

Higher scores are better!

Document Location metric

- Document location means the location of the words in the search query on the page
- It builds on the idea that if a word is relevant for the page, it appears close to the top of that page

Document Location metric

```
void document_loc(Page p, String query)
  //Split search query to get each word
  String[] qws = query.split(" ")
  double score = 0
  //Iterate over each word in the search query
  foreach (String q : qws)
    //Iterate over all words in the page
    boolean found = false
    for (i = 0 to p.words().length())
      String word = p.words().get(i)
      //Score is the index of the first occurence of the
      //word + 1 (to avoid zero scores)
      if (word == q)
        score += i + 1
        //Stop once the word has been found
        found = true
        break
    //If the word is not found on the page, increase
    //the score by a high value
    if (!found)
      score += 100000
  //Return the score
  return score
```


Now, lower scores are better!

Word Distance metric

- Word Distance is based on the idea that queries using multiple words often find results more relevant if the words appear close together on the page
- The metric therefore uses the distance between pair of words in the document
- It can only be used if we have two or more words in the query

Word Distance metric

```
void word_dist(Page p, String query)
  //Split search query to get each word
  String[] qws = query.split(" ")
  double score = 0
  //Iterate over each pair if words in the search query
  for (i = 0 to qws.length() - 1)
    //Use the document location function to get the
    //location of the words
    int loc1 = document_loc(p, qws[i])
    int loc2 = document_loc(p, qws[i+1])
    //Increase the score by the distance between the two words,
    //or a high value if any word is not found in the page
    if (loc1 == 100000 or loc2 == 100000)
      score += 100000
    else
      score += Abs(loc1 – loc2)
  //Return the score
  return score
```

Again, lower scores are better!

Which metric to use?

- There is no universal metric that consistently give good results
- We therefore often have to combine different metrics
- We can also experiment with giving different weights to different metrics
- The total score can for example be calculated as (assuming metric scores are normalized):

totalScore = 1.0 * WordFrequency + 0.8 * DocumentLocation + 0.5 * WordDistance

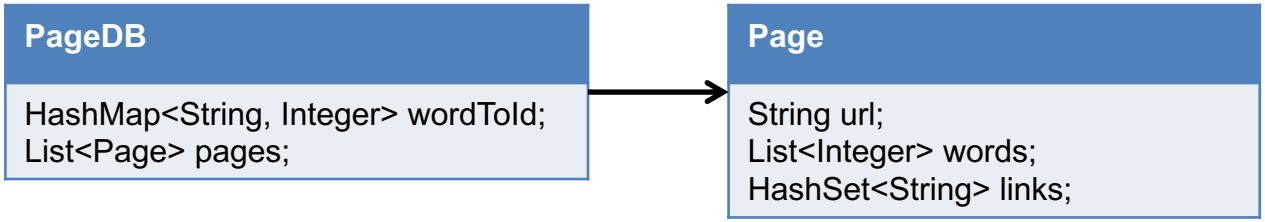
Order by links

Inbound-Link ranking

- The second approach we will use is *Inbound-Link* ranking
- It is different from *Content-Based* ranking in that it doesn't use the contents of a page
- Instead it uses information others have provided about a page, more specifically who has linked to the page
- The idea is that bad pages are less likely to be linked to, and pages with good content have numerous other pages linking to them

Inbound-Link ranking

- To do this we need to slightly modify the code structure we previously has defined.
- Each Page object must now also store all outgoing links from that page:



- Links are stored in a separate file in the Wikipedia data set zip file

Inbound-Link ranking

- We will look into two ways of doing Inbound-Link ranking:
  - Simple Count
  - PageRank algorithm

Simple Count

- As the name implies, this metric is simply a count of how many other pages that link to the current page
- It is easy to find
- For each page, we iterate over all other pages and increase the score by 1 if the other page links to the current page
- This is how academic papers are often ranked.
- A paper is more important if many other papers reference to it

Simple Count

- Using inbound links as the only metric is of course not useable
- It will not care about the search query, it only returns the page with most links to it
- It must be combined with *content-based* ranking
- A drawback with *Simple Count* is that it treats every inbound link equally
- Ideally, we would like to weight inbound links from high quality web sites higher
- This is dealt with in the *PageRank* algorithm

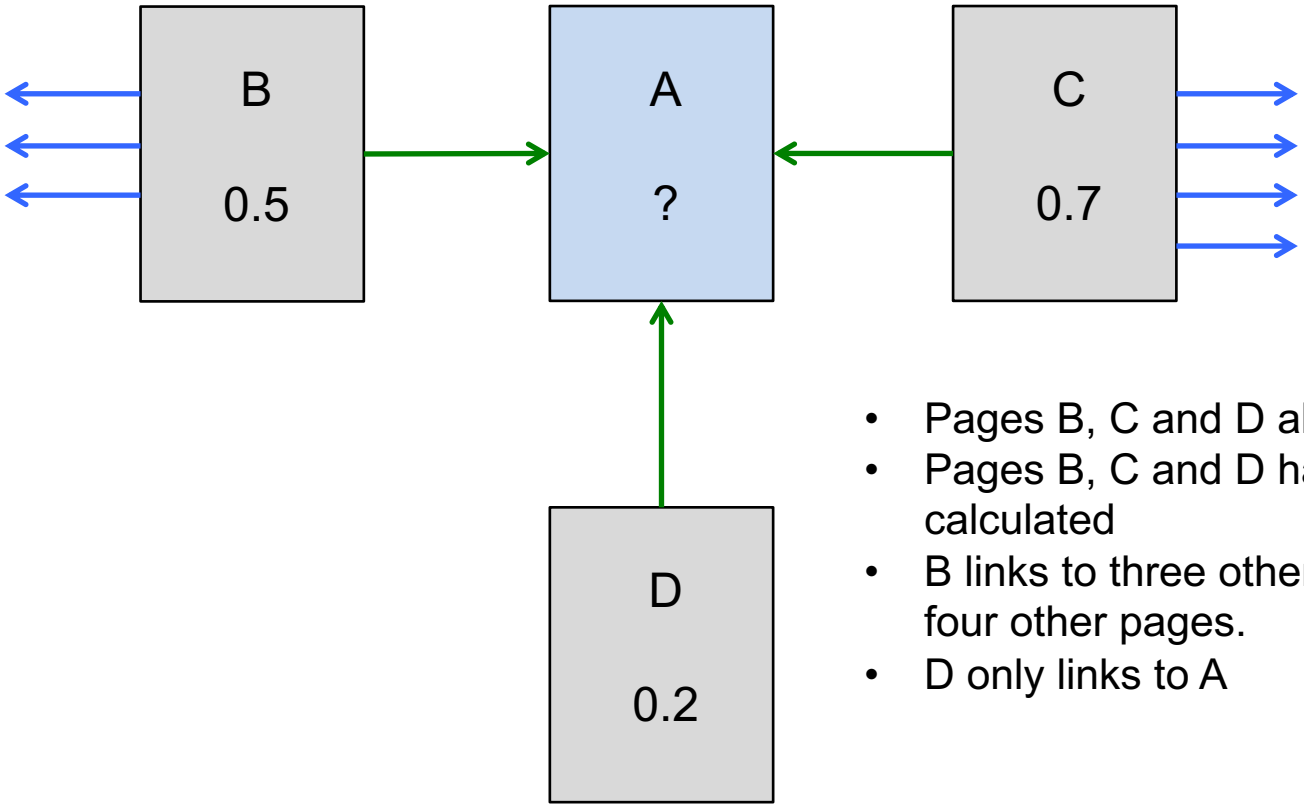
PageRank algorithm

- The PageRank algorithm was first invented by the founders of Google, and is named after Larry Page
- Variations of it are now used by all large search engines
- The basic idea is that every page is assigned a score that indicates how important the page is
- The importance of a page is calculated from the importance of all other pages linking to it
- The importance score is then used to weight inbound links to a page

The Theory

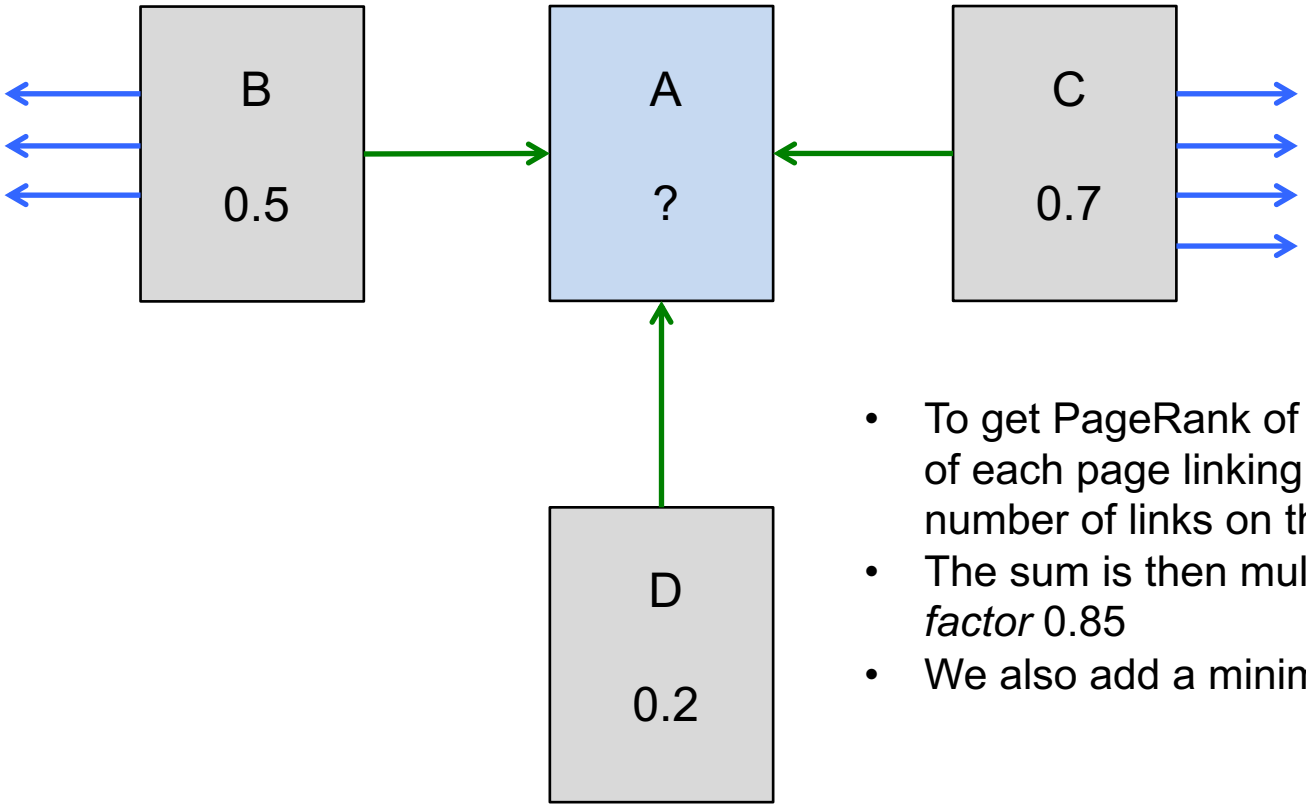
- The algorithm calculates the probability that someone randomly clicking on links will arrive at a certain page
- The more inbound links a page has from other popular pages, the more likely it is that someone will visit the page by pure chance
- If the user keeps clicking forever he will eventually reach every page
- Since users stop surfing after a while, PageRank has a *damping factor* of 0.85 indicating that there is 85% chance that a user will continue clicking from a page

PageRank example



- Pages B, C and D all link to page A
- Pages B, C and D have PageRank values calculated
- B links to three other pages, and C links to four other pages.
- D only links to A

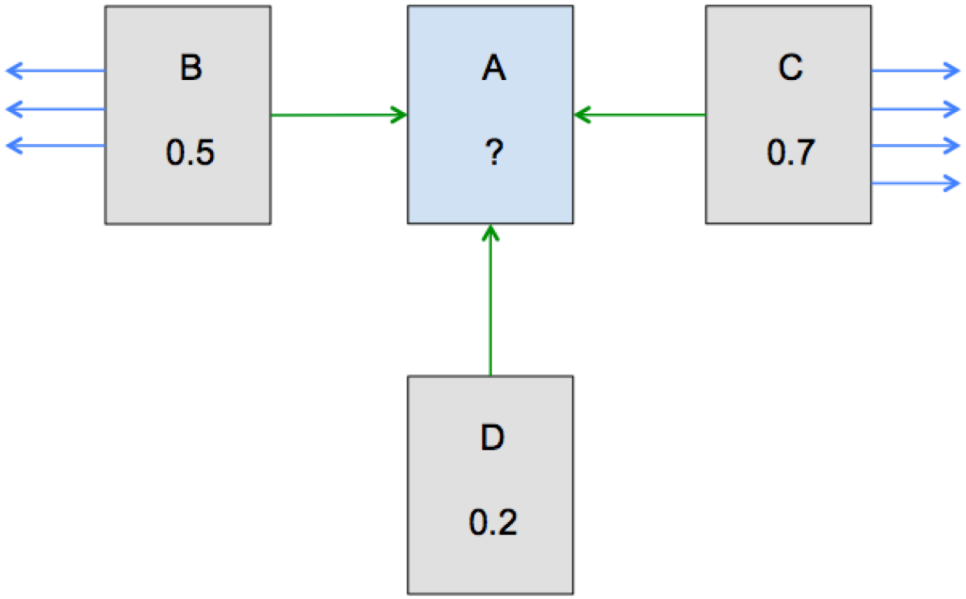
PageRank example



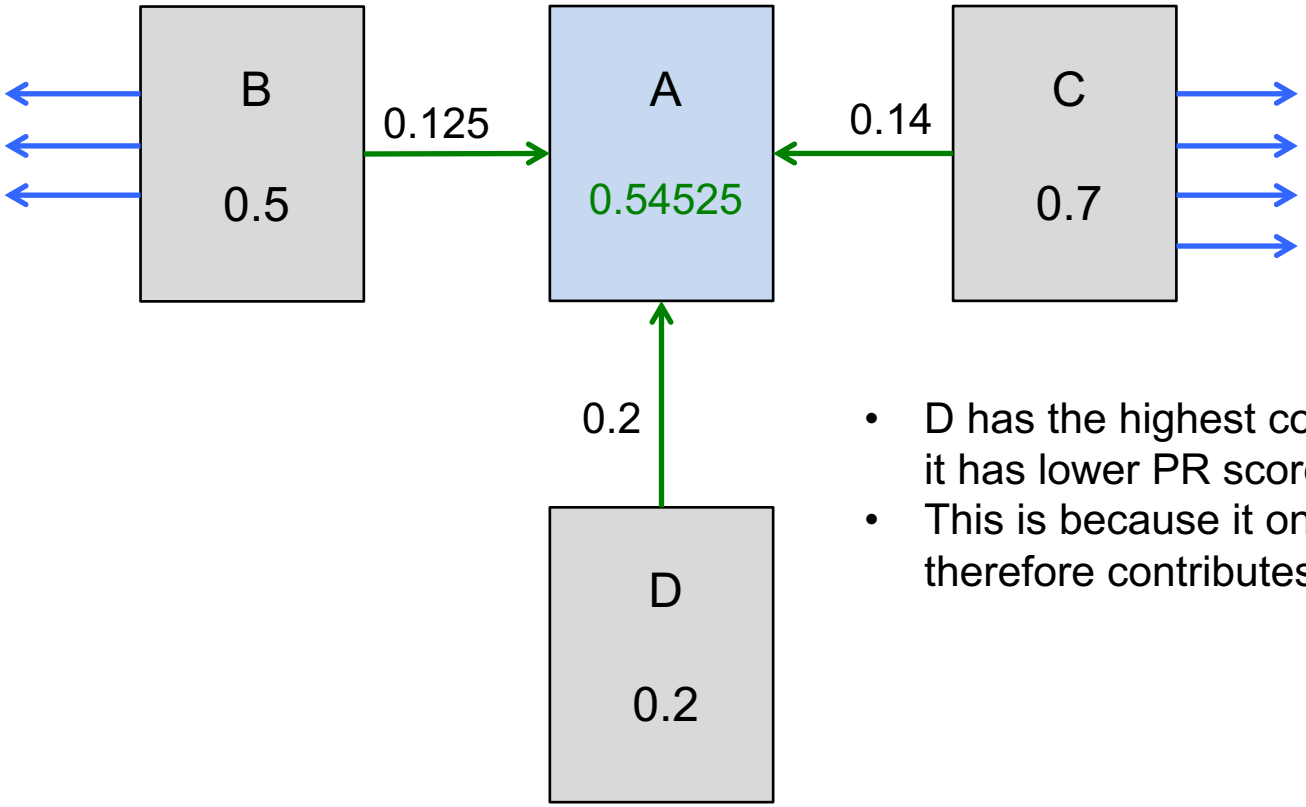
- To get PageRank of A, we take the PageRank of each page linking to A divided by the total number of links on that page
- The sum is then multiplied by the *damping factor* 0.85
- We also add a minimum value of 0.15

PageRank example

PR(A) = 0.15 + 0.85 * (PR(B) / links(B) + PR(C) / links(C) + PR(D) / links(D))
= 0.15 + 0.85 * (0.5/4 + 0.7/5 + 0.2/1)
= 0.15 + 0.85 * (0.125 + 0.14 + 0.2)
= 0.15 + 0.85 * 0.465
= 0.54525



PageRank example



- D has the highest contribution to A even if it has lower PR score than B and C
- This is because it only links to A, and therefore contributes all its score to A

A problem

- The calculation of the PR score for a page is pretty straightforward
- In our example we already knew the PR scores of all pages linking to A
- But what if you didn’t know the PR score of B?
- … and you can’t calculate the score of B without knowing the scores of all pages linking to B?
- For a new set of pages you don’t know any PR score
- How can this be solved?

The solution

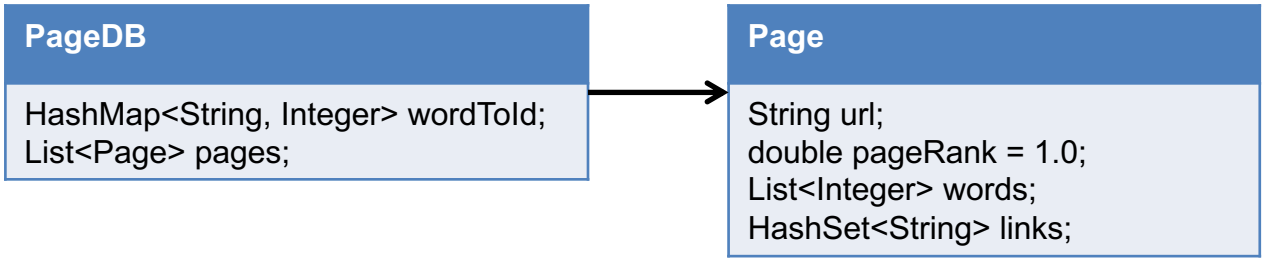
- The solution is to set the PR score to an initial value, for example 1.0
- The PR calculation for each page is then calculated over several iterations
- After each iteration, the PR score for each page gets closer to its true PR score
- The number of iterations needed depends on the number of pages in the set
- In our case we have a small set, so 20 iterations should be enough

Performance considerations

- The PageRank algorithm is time-consuming since we need several iterations to get close to the true PR scores
- Fortunately, we can pre-compute the PR scores for every page
- This is possible because inbound-links ranking does not care about the contents of a page
- Therefore, the PR score is the same no matter what the search query is

PageRank implementation

- First, we need to add a PageRank score to the Page class:



- Then, we need to iterate over a number of iterations, updating PR scores for all pages each iteration

- **Note!** Using a hash set for the links instead of a list improves performance roughly by a factor of 10

PageRank algorithm

```
//Iterate over all pages for a number of iterations
void calculatePageRank()
  for (i = 0 to i < MAX_ITERATIONS)
    //Calculate pagerank values for all pages
    ranks = List()
    foreach (j = 0 to pageDB.pages().length())
      ranks[j] = iteratePR(p)
    //Set new pagerank values for all pages
    foreach (j = 0 to pageDB.pages().length())
      pageDB.pages().get(j).pageRank = ranks[j]


//Calculate page rank value for a page
void iteratePR(Page p)
  double pr = 0
  //Iterate over all pages
  foreach (Page po : pageDB.pages())
    //Check if the other page links to this page
    if (po.hasLinkTo(p))
      //If it does, increase score
      pr += po.pageRank / po.getNoLinks()
  //Calculate and return PR
  pr = 0.85 * pr + 0.15
  return pr
```


Normalization

- The purpose of normalization is to make sure all ranking terms in the score calculation are between 0 and 1
- This makes it easier to find good weights for the different ranking terms
  - *Word Frequency*, *Document Location*, …
- You shall therefore normalize PageRank as well
- This can however be done once after the PageRank update iterations, since the scores are not affected by the search query

Testing it

- The Wikipedia data set consists of 400 pages about programming and 250 pages about video games
- The following metrics and weights are used for ranking the results:

  score = 1.0 * *WordFrequency* + 0.8 * *DocumentLocation* + 0.5 * *PageRank*

- We can try some search queries in our search engine:

Testing it

Search query: nintendo Search

Link	Score	Content	Location	PageRank
Nintendo	2.02	1.00	0.80	0.22
Nintendo_Switch	1.50	0.51	0.80	0.19
Nintendo_Entertainment_System	1.45	0.45	0.80	0.20
List_of_Game_of_the_Year_awards	0.91	0.72	0.00	0.18
Super_Nintendo_Entertainment_System	0.77	0.18	0.40	0.19

Found 111 results in 0.002 sec

Testing it

Search query: java programming Search

Link	Score	Content	Location	PageRank
Java_(programming_language)	2.05	1.00	0.80	0.25
Programming_language	1.26	0.86	0.01	0.38
Object-oriented_programming	0.87	0.46	0.08	0.33
Edsger_W._Dijkstra	0.85	0.57	0.00	0.28
Programming_paradigm	0.78	0.43	0.03	0.32

Found 438 results in 0.004 sec

Testing it

Search query: C++ Search

Link	Score	Content	Location	PageRank
C++	2.04	1.00	0.80	0.24

Found 1 results in 0.002 sec

Notes on performance

- Selecting suitable data structures are necessary to get good performance
- Using a HashSet with links instead of a List reduced the time needed to calculate PageRank from around 50 sec to around 4 sec on my laptop

- Are there other performance improvements you can think of?

Search Engines

Dr. Johan Hagelbäck

johan.hagelback@lnu.se

http://aiguy.org