Web Search

COMP90049
Knowledge Technologies

Rao Kotagiri and Jeremy Nicholson and Justin Zobel, CIS

Semester 1, 2018

THE UNIVERSITY OF MELBOURNE

Elements of a web search engine

Web search involves four main technological components.

- **Crawling**: the data to be searched needs to be gathered from the web.
- **Parsing**: the data then needs to be translated into a canonical form.
- **Indexing**: data structures must be built to allow search to take place efficiently.
- **Querying**: the data structures must be processed in response to queries.

Practical search also involves an increasingly wide range of 'add-on' technologies, such as:

- Snippet generation.
- As-you-type querying.
- Query correction.
- Answer consolidation. (cf. Product price lists)
- Info boxes. (cf. Google Knowledge Graph)

Crawling fundamentals

Before a document can be queried, the search engine must know that it exists. On the web, this is achieved by *crawling*.

(Web crawlers are also known as spiders, robots, and bots.)

Crawlers attempt to visit every page of interest and retrieve them for processing and indexing.

Basic challenge: there is no central index of URLs of interest.

Secondary challenges:

- Some websites return the same content as a new URL at each visit.
- Some pages never return status 'done' on access.
- Some websites are not intended to be crawled.
- Much web content is generated on-the-fly from databases, which can be costly for the content provider, so excessive numbers of visits to a site are unwelcome.
- Some content has a short lifespan.
- Some regions and content providers have low bandwidth.

Crawling

The observation that allows effective harvesting of the web is that it is a highly linked graph.

Assumption: if a web page is of interest, there will be a link to it from another page.

Corollary: given a sufficiently rich set of starting points, every interesting site on the web will be reached eventually.

In principle:

1. Create a prioritised list L of URLs to visit, and a list V of URLs that have been visited and when.
2. Repeat forever:
   1. Choose a URL u from L and fetch the page $p(u)$ at location u.
   2. Parse and index $p(u)$, and extract URLs $\{u'\}$ from $p(u)$.
   3. Add u to V and remove it from L. Add $\{u'\} - V$ to L.
   4. Process V to move expired or 'old' URLs to L.

In practice, page processing is much faster than URL resolution, so numerous streams of pages should be processed simultaneously.

Challenges

The list of URLS L must be prioritised to ensure that

- Every page is visited eventually.
- Synonym URLs are disregarded.
- Significant or dynamic pages are visited sufficiently frequently.
- The crawler isn't cycling indefinitely in a single web site (caught in a crawler trap).

Crawler traps are surprisingly common. For example, a 'next month' link on a calendar can potentially be followed until the end of time.

The Robots Exclusion Standard defines a protocol that all crawlers are supposed to observe. It allows website managers to restrict access to crawlers while allowing web browsing.

Simple crawlers are now part of programming languages, for example Perl's `LibWWW`, and good crawlers are available as part of systems such as `Nutch`.

Page recognition

Once a document has been fetched, it must be *parsed*.

That is, the words in the document are extracted, then added to a data structure that records which documents contain which words.

At the same time, information such as links and anchors can be analysed, formats such as PDF or Postscript or Word can be translated, the language of the documents can be identified, and so on.

First step: determining the format of the page.

The most basic element is the character encoding, which has to be captured in the page's metadata.

- For the first decade or so of the web, most pages were in ASCII. (Want to travel in time? Try the **Wayback Machine**. `waybackmachine.org/19970501000000*/http://cs.mu.oz.au`)
- HTML markup was used to provide an extended character set.
- ISO-8859 and ISO-8859-* now provide extended Latin character sets (Cyrillic, Thai, Greek, . . .)
- UTF-8 is the dominant character set covering the large-alphabet languages, with codes from 8 to 32 bits. The first 128 of the 8-bit codes are ASCII.

Page analysis

Web pages are supposed to be in HTML or XML (or sometimes in other formats, hence `ftp://` and so on).

The format separates user-visible content from metadata.

In most cases, search engine designers actively seek to avoid indexing invisible content; it misleads users and allows spoofing. Thus metadata is generally not a key component of search. (Another form of spoofing is use of tricks such as white text on a white background.)

Many, many websites are not in conformant HTML or XML. Errors can be accidental, or can be a deliberate attempt to take advantage of known behaviour of particular browsers.

Parsers therefore need to be robust and flexible.

Some applications also make use of *scraping*, where only some components of the page are retained. For example, the advertisements and comments on a blog website might be ignored, with only blog content retained for indexing.

Tokenisation

Tokenisation

```
<head>
<META NAME="keywords" CONTENT="science humor, science humour, science,
humor, humour, ig-nobel, ig nobel, ignobel, hotair, hot-air, hot air,
improbable research">
<META HTTP-EQUIV="expires" CONTENT="0">
<title>HotAIR - Rare and well-done tidbits from the Annals of
Improbable Research</title>
</head>

<a href="/navstrip/about.html">About AIR</a>
| <a href="/navstrip/subscribe.html"><font color="red">Subscribe</font></a>
| <a href="http://improbable.typepad.com/">Our blog </a>
| <a href="/navstrip/schedule.shtml">Events calendar</a>
| <a href="/navstrip/contact.html">Contact us</a>
| <a href="/navstrip/google-search.html">Search</a>
<hr>

<img src="/toplevel/banner-2004.gif" width="406" height="200"
alt="The Annals of Improbable Research: HotAIR">

<tr>
<td colspan=2 align="center">
<b><br><b>NOTE THIS:  JoAnn O'Linger-Luscusk and Alasdair
Skelton have <a href="/projects/hair/hair-club-top.html#newest">joined</a>
the Hair Club</b>
</td> </tr>
```

hotair rare and well done tidbits from the annals of improbable research note this joann o linger luscusk and alasdair skelton have joined the hair club

Tokenisation

The aim of parsing is to reduce a web page, or a query, to a sequence of *tokens*.

If the tokenisation is successful, the tokens in a query will match those of the web page, allowing query evaluation to proceed without any form of approximate matching.

Documents typically consist of reasonably well-formed sentences, allowing effective parsing and resolution of issues such as (in English):

- Hyphenation. Is 'Miller-Zadek' one word or two? Is 'under-coating' one word or two? 'Re-initialize'? 'Under-standing'?
- Compounding. Is 'football' one word or two? 'Footballgame'?
- Possessives. Is 'Zadek's' meant to be 'Zadek' or 'Zadeks'? What about 'Smiths'?

Sometimes it is possible to disambiguate word senses, for example to separate 'listen to the wind' from 'wind up the clock', but in practice the error rate obliviates any possible gains.

In any case, such corrections are typically difficult or impossible in queries.

Canonicalisation

Any indexing process that relies on fact extraction may need information in a canonical form.

- Dates. Consider 5/4/2011, 4/5/2011, April 5 2011, first Tuesday in April 2001.
- Numbers. 18.230,47 versus 18,230.47. Or 18 million versus 18,000,000.
- Variant spelling. Color versus colour.
- Variant usage. Dr versus Doctor. (What is the top match for `Dr Who` under Google?)
- Variant punctuation. ‘e.g.’ versus ‘eg’.

Historically, search engines discarded both stop words (‘content-free’ terms such as `the`, `or`, and so on), but they now generally appear to be indexed.

They also discarded terms that linguistic rules suggested were not reasonable query strings, but anecdotally it is reported that they index *all* tokens of up to 64 characters.

Stemming

The most significant form of canonicalisation (for English) is arguably stemming.

This are an attempt to undo the processes that lead to word formation. Most words in English are derived from a *root* or *stem*, and it is this stem that we wish to index, rather than the word itself.

Inflectional morphology: how a word is derived from a stem, for example *in+expense+ive* → *inexpensive*.

Stemming is the process of stripping away affixes.

It can be challenging, because every word has a different set of legal suffixes.

Stemming

Different stemmers have different strengths, but the Porter stemmer (`www.tartarus.org/~martin/PorterStemmer` has several implementations) is the most popular.

It is implemented as a cascaded set of rewrite rules such as

- `sses` → `ss`
- `ies` → `i`
- `ational` → `ate`
- `tional` → `tion`
- `tion` → `-`

Some versions of the stemmer constrain it so that the final result, or the stem produced at each step, must be a known word (either in a dictionary, or in the corpus being indexes).

Stemming example

`glasses` → `glass`

`companies` → `compani`

`positional` → `position` → `posi`

`posies` → `posi`

Other alternatives, like **lemmatisation** stop once we arrive at a dictionary entry, and constrain intermediate steps to dictionary entries

Zoning

Web documents can usually be segmented into discrete zones such as title, anchor text, headings, and so on.

Parsers also consider issues such as font size, to determine which text is most prominent on the page and thus generate further zones.

Web search engines typically calculate weights for each of these zones, and compute similarities for documents by combining these results on the fly.

Hence the observed behaviour of web search engines to favour pages that have the query terms in titles.

Indexing

Fast query evaluation makes use of an *index*: a data structure that maps terms to the documents that contain them. For example, the index of a book maps a few key terms to page numbers.

With an index, query processing can be restricted to documents that contain at least one of the query terms.

Many different types of index have been described.

The only practical index structure for text query evaluation is the *inverted index*: a collection of lists, one per term, recording the identifiers of the documents containing that term.

An inverted index can be seen as the transposition of document-term frequency matrix accessed by (d, t) pairs into one accessed by (t, d) pairs.

Inverted index components

Search structure

For each distinct word t, the search structure contains:

- A pointer to the start of the corresponding inverted list.
- A count f_t of the documents containing t.

That is, the search structure contains the *vocabulary*.

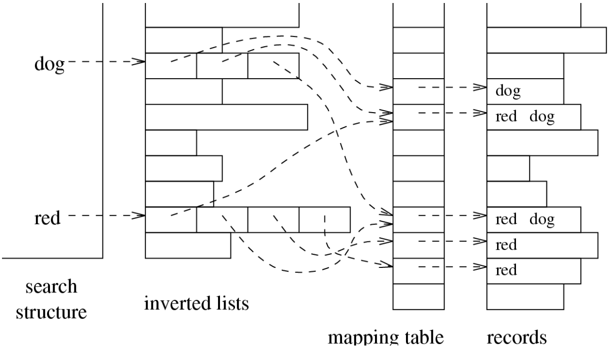
Inverted lists

For each distinct word t, the inverted list contains:

- The identifiers d of documents containing t, as ordinal numbers.
- The associated frequency $f_{d,t}$ of t in d. (We could instead store $w_{d,t}$ or $w_{d,t}/W_d$.)

Example inverted index

Together with an array of W_d values (stored separately), the search structure and inverted index provide all the information required for Boolean and ranked query evaluation.



Example inverted index

For example:

We few, we happy few, we band of brothers

$$\langle \text{a}, \text{aardvark}, \ldots, \text{band}, \ldots, \text{brothers}, \ldots, \text{few}, \ldots, \text{happy}, \ldots \rangle$$

$$\langle 0, 0, \ldots, 1, \ldots, 1, \ldots, 2, \ldots, 1, \ldots \rangle$$

Example inverted index

Inverted index (one document):

band	$\rightarrow$	1(1)
brothers	$\rightarrow$	1(1)
few	$\rightarrow$	1(2)
happy	$\rightarrow$	1(1)
of	$\rightarrow$	1(1)
we	$\rightarrow$	1(3)

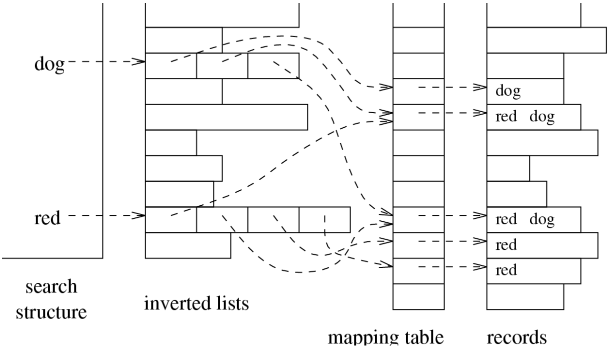
Example inverted index

Inverted index (multiple documents):

$$
\begin{array}{llllll}
\ldots \\
\text{band} & \rightarrow \ldots & \rightarrow & (d, f_{d,\text{band}}) & \rightarrow & \ldots \\
\ldots \\
\text{brothers} & \rightarrow \ldots & \rightarrow & (d, f_{d,\text{brothers}}) & \rightarrow & \ldots \\
\ldots \\
\text{few} & \rightarrow \ldots & \rightarrow & (d, f_{d,\text{few}}) & \rightarrow & \ldots \\
\ldots \\
\text{happy} & \rightarrow \ldots & \rightarrow & (d, f_{d,\text{happy}}) & \rightarrow & \ldots \\
\ldots \\
\text{of} & \rightarrow \ldots & \rightarrow & (d, f_{d,\text{of}}) & \rightarrow & \ldots \\
\ldots \\
\text{we} & \rightarrow \ldots & \rightarrow & (d, f_{d,\text{we}}) & \rightarrow & \ldots \\
\ldots
\end{array}
$$

Example inverted index

An inverted index allows for fast querying because:
(1) the terms in the query correspond to the search structure
(2) the index only indicates documents where the term is *present*



Inverted index size

In a simple representation, for (say) a gigabyte of newswire data,

- 12 MB (say) for 400,000 words, pointers, counts.
- 280 MB for 70,000,000 document identifiers (4 bytes each).
- 140 MB for 70,000,000 document frequencies (2 bytes each).

The total size is 432 MB, or just over 40% of the original data.

For 100 GB of web data, the total size is about 21 GB, or just over 20% of the original text. (Many web pages contain large volumes of unindexed data such as markup.)

Index construction and index maintenance – beyond the scope of this subject. But it is straightforward to build an index for a terabyte of text data on a current laptop in about a day.

Boolean Querying using a TDM

A term–document matrix of binary values is compact to store (1b per term per document), and the bitwise comparisons are fast to perform.

Consequently, a tailored TDM for Boolean querying is preferable over modest document collections: hundreds of thousands of documents implies a matrix of hundreds of MB, which will fit in main memory (not cache); hundreds of millions of documents implies that the matrix is much larger.

Also, most of the values in the matrix are 0, which means that there are many, many comparisons for documents that don't contain any part of the query.

Boolean Querying using an inverted index

To evaluate a general Boolean query using an inverted index,

- Fetch the inverted list for each query term.
- Use intersection of lists to resolve AND.
- Use union of lists to resolve OR.
- Take the complement of a list to resolve NOT (how?).
- Ignore within-document frequencies.

For strictly conjunctive queries, query processing should start with the shortest list as a set of *candidates*, and then eliminate documents that do not appear in the other lists, working from second shortest to longest.

Ranked Querying principles

To produce a document ranking for a typical TF-IDF model, using the cosine as a similarity measure, we need the following information:

- The frequency of each query term in each document (TF)
- The number of documents where each query term occurs (DF)
- The length of each document

Typical cosine:

$$S(q, d) = \frac{q \cdot d}{|q||d|}$$

We then calculate the dot product, and then divide by the vector lengths.

A TDM (32 bits per term per document) is too large to contemplate.

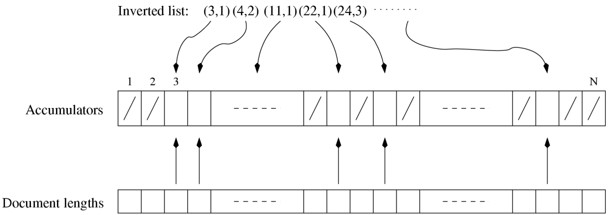
The structure of the inverted index is not designed to compare documents one at a time.

Ranked Querying using an inverted index

To use an inverted index to evaluate a query under the cosine measure,

1. Allocate an accumulator A_d for each document d, and set $A_d \leftarrow 0$.
2. For each query term t,
   1. Calculate $w_{q,t}$, and fetch the inverted list for t.
   2. For each pair $\langle d_t, f_{d,t} \rangle$ in the inverted list
      Calculate $w_{d,t}$, and
      Set $A_d \leftarrow A_d + w_{q,t} \times w_{d,t}$.
3. Read the array of W_d values and, for each $A_d > 0$,
   Set $A_d \leftarrow A_d / W_d$.
4. Identify the r greatest A_d values and return the corresponding documents.

Ranked Querying using an inverted index ...



That is, starting with a set of N zero'ed accumulators, use the lists to update the accumulators term by term.

Then use the document lengths to normalize each non-zero accumulator.

Accumulator costs

With the standard query evaluation algorithm and long queries, most accumulators are non-zero and an array is the most space- and time-efficient structure.

But the majority of those accumulator values are trivially small, with the only matching terms being one or more common words. And note that the accumulators are required on a per-query basis.

If only low f_t (that is, rare) terms are allowed to create accumulators, the number of accumulators is greatly reduced.

A simple mechanism is to impose a limit L on the number of accumulators. This is another example of an efficiency-driven compromise that alters the set of documents returned, and may therefore impact on effectiveness.

The “limiting” approach

1. Create an empty set A of accumulators.
2. For each query term t, ordered by decreasing $w_{q,t}$
   1. Calculate $w_{q,t}$, and fetch the inverted list for t.
   2. For each pair $\langle d_t, f_{d,t} \rangle$ in the inverted list
      If there is no accumulator for d and $|A| < L$,
      create an accumulator A_d for d.
      If d has an accumulator calculate $w_{d,t}$ and
      set $A_d \leftarrow A_d + w_{q,t} \times w_{d,t}$.
3. For each accumulator set $A_d \leftarrow A_d / W_d$.
4. Identify the r greatest A_d values and return these documents.

There are many variations on these algorithms.

The “thresholding” approach

1. Create an empty set A of accumulators, and set a threshold S.
2. For each query term t, ordered by decreasing $w_{q,t}$
   1. Calculate $w_{q,t}$, and fetch the inverted list for t.
   2. For each pair $\langle d_t, f_{d,t} \rangle$ in the inverted list
      Calculate $w_{d,t}$.
      If there is no accumulator for d and $w_{q,t} \times w_{d,t} > S$,
      create an accumulator A_d for d.
      If d has an accumulator
      set $A_d \leftarrow A_d + w_{q,t} \times w_{d,t}$.
3. For each accumulator set $A_d \leftarrow A_d / W_d$.
4. Identify the r greatest A_d values and return these documents.

Querying costs

Several resources must be considered.

Disk space: for the index, at 40% of the size of the data. (With unstemmed terms, the index can be around 80% of the size of the data.)

Memory space: for accumulators, for the vocabulary, and for caching of previous results.

CPU time: for processing inverted lists and updating accumulators.

Disk traffic: to fetch inverted lists.

By judicious use of compression and careful pruning, all of these costs can be dramatically reduced compared to this first implementation. The gains are so great that it makes no sense to implement without some use of compression.

Phrase queries

Around 7% of the queries in the Excite log have an explicit phrase, such as `"the great flydini"`.

Also, around 43% evaluate successfully if treated as a phrase, that is, the words must be adjacent in the retrieved text. People enter phrases without putting quotes in. It makes sense to give such pages a higher score than pages in which the words are separated.

A question for information retrieval research (and outside the scope of this lecture) is how best to use phrases in similarity estimation.

A question for research in efficient query evaluation is how to find the pages in which the words occur as a phrase.

Phrase queries

The number of distinct phrases grows far more rapidly than the number of distinct terms. A small web crawl could easily contain a billion distinct two-word pairs, let alone longer phrases of interest.

There are three main strategies for phrase query evaluation:

- Process queries as bag-of-words, so that the terms can occur anywhere in matching documents, then post-process to eliminate false matches.
- Add word positions to the index entries, so the location of each word in each document can be used during query evaluation.
- Use some form of phrase index or word-pair index so that they can be directly identified without using the inverted index.

In this lecture, inverted lists have been described as a sequence of index entries, each an $\langle d, f_{d,t} \rangle$ pair. It is straightforward to include the $f_{d,t}$ ordinal word positions p at which t occurs in d:

$$\langle d, f_{d,t}, p_1, \ldots, p_{f_{d,t}} \rangle$$

Positions are word counts, not byte counts, so that they can be used to determine adjacency.

Phrase query evaluation

A phrase in a ranked query can be treated as an ordinary term – a lexical entity that occurs in given documents with given frequencies.

Similarity can therefore be computed in the usual way, but it is first necessary to use the inverted lists for the terms in the phrase to construct an inverted list for the phrase itself.

This requires that the index be extended to include word positions in each document, along with in-document frequency.

- Fetch the inverted lists for each term.
- Take their intersection to find locations at which the phrase occurs.

A similar strategy can be used for the more general task of determining whether query terms are proximate in a document.

Phrase query evaluation

Many phrases include common words. The cost of phrase query processing on an inverted index is dominated by the cost of fetching and decoding lists for these words, which typically occur at the start of or in the middle of a phrase.

These words could be neglected. For example, evaluation of the query

```
the lord mayor of melbourne
```

could involve intersecting the lists for `lord`, `mayor`, and `melbourne`, looking for positions p of `lord` such that `mayor` is at $p + 1$ and `melbourne` is at $p + 3$.

False matches could be eliminated by post-processing, or could simply be ignored.

Phrase query evaluation ...

Alternatively, it is straightforward to build a complete index of two-word phrases (around 50% of the size of the "web" data). Then evaluation of the phrase query:

```
the lord mayor of melbourne
```

involves only lists for, say, the phrases `the lord`, `mayor of`, and `of melbourne`.

Proximity is an a variant, imprecise form of phrase querying.

- Favour documents where the terms are near to each other.
- Search for "phrases" where the terms are within a specified distance of each other.

Proximity search involves intersection of inverted lists with word positions.

Link analysis

In general search, each document in considered independently.

In web search, a strong piece of evidence for a page's importance is given by *links*, in particular how many other pages have links to this page.

(This can be spoofed by use of link farms, but with the kinds of analysis used by current engines it is extremely hard to do so effectively.)

The two major link analysis algorithms are HITS (hyperlinked-induced topic search, not discussed in this subject) and PageRank.

Pagerank overview

Basic intuition of PageRank: each web document has a fixed number of credits associated with it, a portion of which it redistributes to documents it links to; in turn, it receives credits from pages that point to it.

The final number of credits the page is left with determines its pagerank $\pi(d) \in [0, 1]$, where $\sum \pi(*) = 1$.

The process used to calculate the $\pi(d)$ values is based on the notion of "random walks" with the option to 'teleport' to a random page with fixed probability $\alpha \in (0, 1)$. In this, we make the following assumptions:

- Each page has the same probability of being the start point for the random walk.
- For both teleports and traversal of outgoing links, all (relevant) pages have an equal probability of being visited.

Some implementations of PageRank assign a maximum, fixed score to trusted pages, to seed the process.

Pagerank in practice

PageRank has a reputation for being critical to the performance of Google, and has attracted a great deal of research interest.

Analyses of Google searches has shown that in most cases the importance of PageRank is low.

Anchor text, however, is crucial. For example,

- There are many thousands of "aerospace" web pages in the RMIT web site.
- The Aerospace home page only contains the word once.
- About 95% of the within-RMIT 'aerospace' links point to the Aerospace home page.
- Most of the links to the home page contain the word 'aerospace'.

Anchor text is treated as a form of zone.

A high-performance web search engine

Further heuristics.

- Note which pages people actually visit by counting click-throughs.
- Manually alter the behavior of common queries.
- Cache the answers to common queries.
- Index selected phrases.
- Divide the collection among multiple servers, each of which has an index of its documents.
  Then have multiple collections of identical servers.
- Have separate servers for crawling and index construction.
- Accept feeds from dynamic data providers such as booksellers, newspapers, and microblogging sites.
- Integrate diverse data resources, such as maps and directories.

Summary

- Search involves crawling, parsing, indexing, and querying; practical search also involves a range of other technologies.
- Crawling is in principle a straightforward application of queuing, but practical issues mean that implementation is complex.
- Parsing involves discarding metadata and hidden information; tokenization; canonicalisation; zoning; and stemming.
- Inverted indices describe text collections as lists of the pages with each word, rather than the list of words on each page.
- The same structure is used for Boolean and ranked querying.
- Approximations can be used to reduce querying costs, which can affect the answer set in unpredictable ways.
- On the web, link and anchor information can be the dominant evidence of relevance.

Readings

Brin, Sergey and Lawrence Page (1998). “The Anatomy of a Large-Scale Hypertextual Web Search Engine”. Computer Networks 30: 107–117.

Barroso, Luiz André, Jeffrey Dean, and Urs Hötzle (2003) “Web Search for a Planet: The Google Cluster Architecture”. IEEE Micro 23 (2): 22–28. doi:10.1109/MM.2003.1196112

Zobel, Justin and Alistair Moffatt (2006). “Inverted Files for Text Search Engines”. ACM Computing Surveys 38 (2): 1–56. doi:10.1145/1132956.1132959

Manning, Christopher D., Prabhakar Raghavan, Heinrich Schütze (2008). “Introduction to Information Retrieval”. Chapters 1–2, 20–21. Cambridge University Press.

Pagerank algorithm

Input: D = document set
Output: Π_T = set of pagerank scores for each document $d_i \in D$

for all $d_i \in D$ **do** $\triangleright$ Initialise the starting probabilities
    $\pi(d_{(i,0)}) \leftarrow \frac{1}{N}$ $\triangleright$ N is the total number of documents
end for
for $t = 1..T$ **do** $\triangleright$ Repeat over T iterations
    for all $d_i \in D$ **do** $\triangleright$ Initialise the document probabilities
        $\pi(d_{(i,t)}) \leftarrow 0$
    end for
    for all $d_i \in D$ **do**
        if $\exists d_j : d_i \mapsto d_j$ **then**
            for all $d_j \in D$ **do** $\triangleright$ EITHER teleport randomly
                $\pi(d_{(j,t)}) \leftarrow \pi(d_{(j,t)}) + \alpha \times \pi(d_{(i,t-1)}) \times \frac{1}{N}$
            end for
            for all d_j **where** $d_i \mapsto d_j$ **do**
                $\triangleright$ OR follow an outlink (one of m)
                $\pi(d_{(j,t)}) \leftarrow \pi(d_{(j,t)}) + (1 - \alpha) \times \pi(d_{(i,t-1)}) \times \frac{1}{m}$
            end for
        else
            for all $d_j \in D$ **do** $\triangleright$ teleport to a random document
                $\pi(d_{(j,t)}) \leftarrow \pi(d_{(j,t)}) + \pi(d_{(i,t-1)}) \times \frac{1}{N}$
            end for
        end if
    end for
end for

Pagerank example

Assume a set of two documents, d_1 and d_2, with a link from d_1 to d_2.

t	$\pi(d_{(1,t)})$	$\pi(d_{(2,t)})$
0	0.5	0.5
1	$0.5 \times 0.2 \times 0.5 + 0.5 \times 0.5 = 0.3$	$0.5 \times 0.2 \times 0.5 + 0.5 \times 0.8 + 0.5 \times 0.5 = 0.7$
2	$0.3 \times 0.2 \times 0.5 + 0.7 \times 0.5 = 0.38$	$0.3 \times 0.2 \times 0.5 + 0.3 \times 0.8 + 0.7 \times 0.5 = 0.62$
3	$0.38 \times 0.2 \times 0.5 + 0.62 \times 0.5 = 0.348$	$0.38 \times 0.2 \times 0.5 + 0.38 \times 0.8 + 0.62 \times 0.5 = 0.652$

$$\vdots$$