What is Computer Science?

An Information Security Perspective

Daniel Page ⟨dan@phoo.org⟩ and Nigel P. Smart ⟨csnps@bristol.ac.uk⟩

git # b4055dd3 @ 2019-05-13

DEMYSTIFYING WEB-SEARCH: THE MATHEMATICS OF PAGERANK

Ask yourself a question: other **web-search** engines [39] exist of course, but how often do you use Google via

https://www.google.com/

to search for something? In fact, if you have a Google account the answer is available at

https://history.google.com/

unless you turned this feature off. For me it was around 100 times a day, although interestingly it changes a lot depending on what day it is. Another one: how often does the set of results produced *fail* to include what you were searching for? This is harder to answer precisely, but my guess would be not *that* often overall. Even accepting that it might fail sometimes, if you stop to think about it this is really amazing: versus only a generation ago, it seems fair to say that the ability to access so much information with such ease and accuracy has changed the world we live in fundamentally.

Although online advertising underpins the Google business model (a large proportion of income stemming from the **AdWords** [2] and **AdSense** [1] systems), from a technology perspective their web-search system remains a core interest. To support it, Google store and process a *lot* of information (some estimates cite upward of 50 billion web-sites alone, on top of which they deal with images etc.) and deal with a *lot* of **search queries** (40% of the current 7 billion world population make use of the Internet, so if each of them perform 100 Google searches a day we are potentially talking about a huge volume of queries at any given point in time). This means they clearly need a *lot* of computing power to keep pace with demand. Beyond this however, and given all the challenges involved, how are they able to provide such high-*quality* results?

A full answer is obviously quite complicated, and has also changed over time to meet new challenges and capitalise on new opportunities. Even so, the central concepts depend on applying fundamental Computer Science to solve real, practical challenges; better still, they can be explained using Mathematical techniques you are already (somewhat) familiar with. These concepts represent our focus in this Chapter: our aim is to explain how Google produces results for your search queries, using graph theory and probability theory behind the scenes[1].

1 PageRank: the essence of Google web-search

1.1 What actually *is* web-search?

First, a recap of things you probably already know. The **World Wide Web (WWW)** [41] (or **web**) was conceived in a 1989 proposal by Tim Berners-Lee: the idea was to develop a universal mechanism to view (and edit) documents, with embedded connections that would facilitate browsing through them. Overviewing the rich body of previous and related work is beyond our scope, but the proposal basically aimed to implement what

[1] The Chapter assumes you have at least *some* exposure to these topics, and focuses on explaining how they are used. However, we include a number of fairly lengthy introductions in case you need a refresher or even a place to start learning about them from scratch.

was an established concept called **hypertext** [13]; one famous example[2] is the so-called "mother of all demos" in 1969 by Doug Engelbart [32], which included use of the first mouse to navigate hypertext documents in NLS [23]. This aim was achieved when Berners-Lee and a group of collaborators eventually developed

1. the **HTML** [14] language for describing document content, which we now know as **web-pages**, and the embedded **hyperlinks** [11],
2. the **HTTP** [15] protocol used to communicate such content to and from client (or **web-browser**) and **web-server** software, plus
3. the first actual implementations of such software that could be used to form a working system.

The result was advertised[3] to the Internet at large in 1991, and the rest, as they say, is history: various aspects of the original system have matured and improved, but either way it now represents a ubiquitous presence in modern life.

If you think of the web as a massive database of information stored within web-pages, **web-search** can be described as an information retrieval problem [17]: given the database of web-pages, represented by a set V, we want to find the sub-set $R \subset V$ so each $x \in R$ matches a **search query** q. Exactly what constitutes a match depends on the type of search queries allowed [40] of course, so deciding whether a given web-page matches or not is a challenge in itself. To make things simple however, imagine q is just a single word: R might then be the set of web-pages which contain it somewhere.

1.2 Web-search before Google

Depending on how old you are, the birth of the web in 1991 might already seem like a long time ago. But systems for web-search *did* already exist between then and 1998, when Google was founded as a company. Understanding of this history is important, because it explains the technical context into which Google web-search was born a few years earlier (based on a 1996 research project at Stanford University): like most good products it solved a problem, so by understanding the problem means we can better appreciate the solution.

1.2.1 Web-directories

Once a significant number of web-pages were available, the most obvious way to locate one of interest was initially a **web-directory** [37]. Probably the first example, dated roughly 1992, was a list[4] of web-server maintained by Tim Berners-Lee at CERN (when there were few enough to do so easily); contemporary examples include the **WWW Virtual Library** [42], also started by Berners-Lee, at

http://www.vlib.org/

and the **Open Directory Project (ODP)** [24] at

http://www.dmoz.org/

The basic idea is to classify web-pages within a giant table of contents. Much like the table of contents in a book is divided into Chapters and/or Sections, the web-directory is arranged into a hierarchy of categories with all web-pages about a given topic listed under the same category. Imagine some user wants to find web-pages related to some topic q. To do so, they start at the top, least specific level of the hierarchy and move level-by-level through more specific categories until they get to a set R of matches for q. Imagine q = "star wars" for instance: using ODP, they might navigate through

"Top" → "Arts" → "Movies" → "Titles" → "S" → "Star Wars Movies"

until eventually finding (lots of) relevant matches.

1.2.2 Web-search engines

A **web-search engine** (or **search engine**) offers a different way to resolve search queries, basically forming R by testing q against every web-page on behalf of the user. This avoids the need to maintain the hierarchy of categories, and also automates the process so that users no longer need to manually control the search.

A lot of web-search engines have existed, and their history is interesting as a topic in itself. Although it oversimplifies the range of approaches employed somewhat, we can think of them as using three stages:

[2] A copy of the video is preserved at http://www.dougengelbart.org/firsts/dougs-1968-demo.html.
[3] A copy of the post is preserved at http://groups.google.com/groups?selm=6487%40cernvax.cern.ch.
[4] A copy of the web-page is preserved at http://www.w3.org/History/19921103-hypertext/hypertext/DataSources/WWW/Servers.html.

1. **web-crawling** [36],
2. **web-indexing** [35], then
3. search query resolution.

The first and second stages automatically collect and summarise information about web-pages, using the content to form a **web-index** used by the third stage to produce a set of results. The volume of information and relative lack of storage capacity initially limited the information collected to specific features of a web-pages (e.g., within so-called **meta tags** [20], or headings and titles), but this was quickly expanded to full-text search (meaning search of any web-page content).

Think of it like this: the index of a book is basically a map from words (usually those deemed important in some sense), to pages on which they appear. The index is prepared before the book is printed by analysing the content of each pagem, then if you want to know which page includes word x, you look-up x in the index and it tells you. The concept of web-indexing is exactly the same, except we want a mapping from words to *web*-pages. Now, we want to know which web-pages match the search query q, so we look-up q in the web-index and use the associated web-pages as R.

The system as a whole operates in two phases [22]: preparation of the web-index is performed **offline** (meaning before anyone uses the system), while search query resolution is performed **online** (meaning during use of the system). It is important the web-index *can* be prepared offline, because the web-crawling process will take a long time for the entire web. By **pre-computing** [27] it before anyone uses the system, the actual searches queries can still be resolved quickly because the bulk of the work has already been completed.

1.3 Web-search after Google

Arguably, a good web-directory will produce high-quality results provided users can navigate the hierarchy to resolve their query. Why? Ignoring the issue of deciding on a category for web-page x, a quality control system decides whether or not x is good enough to include at all: this decision might be resolved in two parts, by a submitter first suggesting the web-page then an editor making a final decision about inclusion. You could think of this as roughly analogous to them voting for the web-page. However, there are a number of problems, including

- the actual decisions may be subjective and/or hard to articulate precisely, so it is hard to deal with issues such as bias,
- it is difficult to cope with many web-pages, since the workload involved in doing so is relatively high, and
- it sort of assumes the web-pages stay the same, because otherwise the process would have to be repeated again and again, which of course further adds to the workload.

You could also argue that web-search engines solve some of these problems through automation, since the workload can be borne by a computer rather than a human. However, it also seems fair to say they just shift rather than solve the underlying problem: the maintainer of a given web-search engine has an easier task than for a web-directory, but now the *user* is faced with the challenge of enforcing quality control. *They* must filter the good results from potentially lots of bad ones.

PageRank [25] is, roughly speaking, the solution employed by Google. Although it represents an important aspect of their producing such high-quality results, and therefore the success of the company, the concept itself is very easy to grasp: instead of *humans* voting for web-pages, the web-pages will vote for *each other*. Think of each link from web-page x to y as a vote, in the sense that x indicates y has something important (or at least relevant) on it: y is deemed important if many other web-pages link to it, and even more so if those web-pages are themselves important. By considering the *structure* of the web (i.e., the links between web-pages) alongside their content, a PageRank-based web-search engine can solve the quality control problem: it allows us to automatically sort the results, and display the good, important ones before the bad, unimportant ones.

Strictly speaking, PageRank is just a component within the wider Google web-search system: the term may be used to describe either the measure of importance assigned to each web-page, or the algorithm used to compute such values. So given our intuition about what it is, how does it work? This is really three questions in one. In reality, we need to answer the following:

1. how do we collect information to inform our analysis of web-page importance, which is basically the same web-crawling challenge that any other web-search engine faces,
2. how do we formalise the concept of importance, so we can compute actual PageRank values for each web-page, and,

```
1  algorithm BFS(V, E, s) begin
2      Q ← ⟨s⟩, D ← {s}
3      while |Q| ≠ 0 do
4          Remove first element u from Q
5          foreach v such that (u, v) ∈ E do
6              if v ∉ D then
7                  Append v to Q
8                  Add v to D
9              end
10         end
11         Visit and process u
12     end
13     return
14 end
```

(a) *An algorithm for Breadth-First Search (BFS).*

```
1  algorithm DFS(V, E, s) begin
2      Q ← ⟨s⟩, D ← {s}
3      while |Q| ≠ 0 do
4          Remove first element u from Q
5          foreach v such that (u, v) ∈ E do
6              if v ∉ D then
7                  Prepend v to Q
8                  Add v to D
9              end
10         end
11         Visit and process u
12     end
13     return
14 end
```

(b) *An algorithm for Depth-First Search (DFS).*

Figure 1: *Two approaches to traversal of vertices in a graph.*

3. once we have those values, how do we use them while resolving a given search query?

These questions are addressed, in order, by the following Sections.

2 Using graph theory to model and explore the web

As already discussed, the goal of web-crawling is to automatically collect and summarise information about web-pages: we want a computer to do this rather than a human because there are a *lot* of web-pages to process. The idea is to develop an algorithm that controls the web-crawler software, i.e., determines exactly *how* it automatically browses the web. This task can be made a lot easier by using **graph theory**, including various existing algorithms, as a starting point: we model the *real* web as a directed graph

$$G = (V, E)$$

called the **web-graph** [38]: each vertex $v \in V$ represents a web-page, and each edge $(u, v) \in E$ represents a link from web-page u to web-page v. A (very small) example is illustrated by Figure 2, which includes $n = 6$ web-pages identified by URLs such as `a.html`, and links such as from `a.html` to `d.html`. This might not *seem* like a massive step; the web-graph is certainly a natural, and fairly obvious way to model the real web. Crucially though, this added formalism allows us to develop and reason about algorithms that process the web-graph.

2.1 Graph traversal

Given a graph $G = (V, E)$ as input, **graph traversal** [10] is fairly simple problem to explain: starting at some vertex s, the goal is simply to move along edges until all are vertices have been visited. If you translate "vertex"

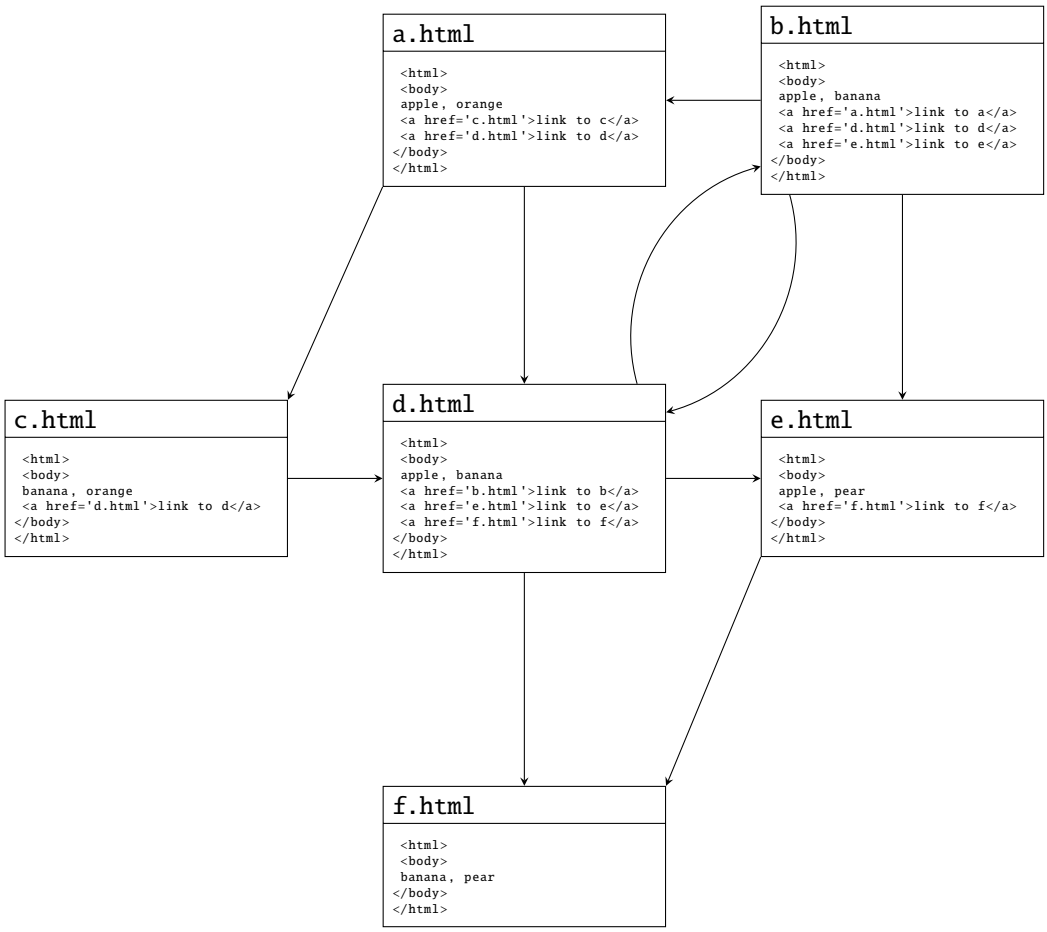


Figure 2: *A simple, concrete web-graph that captures the link structure between, and content of each web-page; each of the $n = 6$ highly artificial web-pages is a short HTML file, which in combination provide structure (i.e., links between the web-pages) and content (i.e., some words, in this case names of fruit).*

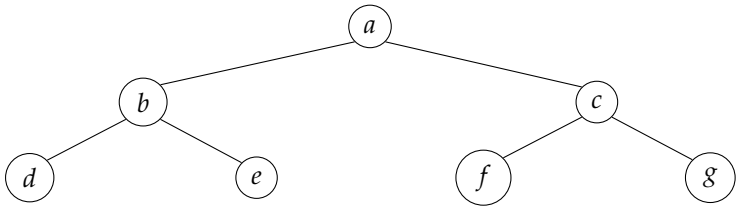
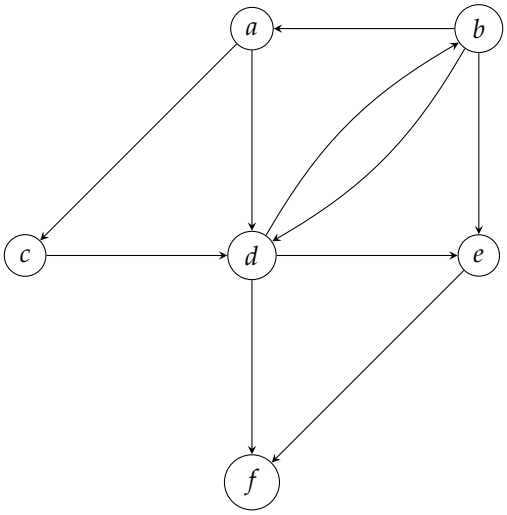
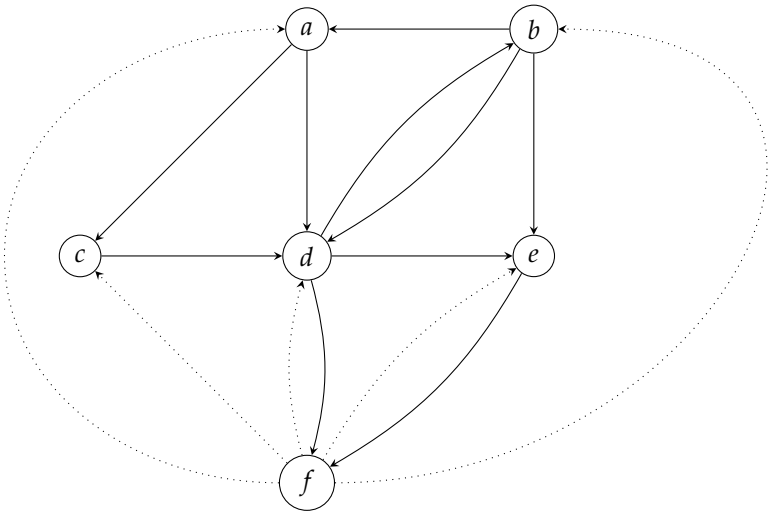


Figure 3: *A simple 7-vertex tree used to explain the behaviour of BFS and DFS algorithms.*



(a) *Before processing to deal with sink web-page f.*



(b) *After processing to deal with sink web-page f; artificially added links from f shown as dashed lines.*

Figure 4: *Two simple, abstract web-graphs (derived from Figure 2) that capture the link structure between, but* not *content of each web-page.*

An aside: a quick introduction to graphs and graph theory.

In **graph theory** [9], a **graph** [8] is used to describe a set of abstract objects and relationships between them. Formally, we specify a graph using two sets:

1. The set V is called the set of **vertices** (or **nodes**): these are basically labels that identify the abstract objects we are interested in.
2. The set E is called the set of **edges**: each edge is a pair of vertices from V, i.e., (u, v) where $u, v \in V$, which specify a relationship between the associated objects.

As a result, we often write $G = (V, E)$ to show that a particular graph G is specified by particular sets V and E. Consider an example where

$$V = \{a, b, c, d\} \qquad E = \{(a, b), (b, d), (d, c), (d, d), (c, a), (c, b)\}$$

From this, we can see there are four vertices labelled a, b, c and d; there are six edges between them in total, meaning for example that a and b are connected and so on. We often visualise the structure of a graph by drawing each of the vertices, and linking them with lines to depict the edges.

Beyond this general definition, several particular types of graph are useful:

Undirected versus directed edges In an undirected graph, each edge simply connects two vertices: if we have (u, v) for instance, this means the same as (v, u) in that both specify an edge between u and v. In a directed graph however, each edge carries some more information. Directed edges still connect vertices, but the order matters. For example, (u, v) is an edge from u to v (but not from v to u) whereas (v, u) is an edge from v to u (but not from u to v). It is common to visualise this using an arrow head on the edge.

Unweighted versus weighted edges In an unweighted graph, the edges carry no information other than the relationship they specify; this alone is enough for many applications. In a weighted graph however, we associated extra information called a **weight** with each edge: a edge between some u and v would look like (u, v, w) where w is the weight, which is usually just a number.

There are numerous properties we can define given some graph G, but an important one is the **degree** of a given vertex $u \in V$: given two functions

$$\begin{aligned} \text{into}(u) &= \{v \mid v \in V, (v, u) \in E\} \\ \text{from}(u) &= \{v \mid v \in V, (u, v) \in E\} \end{aligned}$$

that produce the set of all vertices v where there is an edge from v to u or u to v respectively, the degree of u is $\deg(u) = |\text{from}(u) \cup \text{into}(u)|$ meaning the total number of edges that connect it to any other vertex.

A **path** is a sequence of edges in E which connect a sequence of vertices in V: in a sense, it connects the first vertex in the sequence to the last vertex by moving along the intermediate edges. Within our example, the path $P = \langle (a, b), (b, d), (d, c) \rangle$ connects a to c, for instance; a path like this with no repeated vertices is deemed to be **simple**.

into "web-page" and "visit" into "extract and summarise the content", this should seem reasonably close to what we want.

Algorithm 1a and Algorithm 1b detail two classic ways to solve this problem, called **Breadth-First Search (BFS)** [3] and **Depth-First Search (DFS)** [7] respectively[5]. Each algorithm starts at s, and then visits vertices one at a time by traversing edges from those already visited. To keep track of this process, a sequence Q of vertices yet to visit (called the worklist), and a set D of vertices already visited are maintained. The intuition is as follows:

- The loop in lines #3 to #12 processes vertices until the worklist is empty; line #4 removes the first vertex u from Q and then processes it.
- The loop in lines #5 to #9 process each edge (u, v) from u to some other vertex v: if v has not been visited already (i.e., $v \notin D$), then it is added to both Q and D. Checking D first is important, because it allows **cycles** [5] to be avoided.
- Finally, line #11 visits vertex u, processing it in whatever way is appropriate.

Given a worklist $Q = \langle Q_0, Q_1, \ldots, Q_{n-1} \rangle$, one or two of the steps might need some more detailed explanation:

- In line #7 of Algorithm 1a, where we need to *append* v to Q, we update Q to be $Q \parallel \langle v \rangle = \langle Q_0, Q_1, \ldots, Q_{n-1}, v \rangle$ so that v becomes the new last element.
- In line #7 of Algorithm 1b, where we need to *prepend* v to Q, we update Q to be $\langle v \rangle \parallel Q = \langle v, Q_0, Q_1, \ldots, Q_{n-1} \rangle$ so that v becomes the new first element.
- In line #4 of either algorithm, where we need to remove the first element from Q, we just take Q_0 (i.e., the first element) as u then update Q to be $\langle Q_1, \ldots, Q_{n-1} \rangle$ (i.e., all the other elements).

Beyond this, the behaviour of both algorithms is better explained with an example. Using a **tree** [34], a special type of undirected graph in which any two vertices are connected by one simple path, means their differing behaviours are easier to explain: consider a 7-vertex tree described formally via

$$\begin{aligned} V &= \{a, b, c, d, e, f, g\} \\ E &= \{(a, b), (a, c), (b, d), (b, e), (c, f), (c, g)\} \end{aligned}$$

or visually in Figure 3. If we invoke the BFS algorithm using a (the so-called **root** of the tree) as a starting point, it proceeds as follows:

Step #1 Set $Q = \langle a \rangle$ and $D = \{a\}$.

Step #2 Since $|Q| \neq 0$, perform the next loop iteration for $v \in \{b, c\}$.

- **Step #2.1** Set $u = a$ and $Q = \langle \rangle$.
- **Step #2.2** Since $b \notin D$, set $Q = \langle b \rangle$ and $D = \{a, b\}$.
- **Step #2.3** Since $c \notin D$, set $Q = \langle b, c \rangle$ and $D = \{a, b, c\}$.
- **Step #2.4** Visit a.

Step #3 Since $|Q| \neq 0$, perform the next loop iteration for $v \in \{d, e\}$.

- **Step #3.1** Set $u = b$ and $Q = \langle c \rangle$.
- **Step #3.2** Since $d \notin D$, set $Q = \langle c, d \rangle$ and $D = \{a, b, c, d\}$.
- **Step #3.3** Since $e \notin D$, set $Q = \langle c, d, e \rangle$ and $D = \{a, b, c, d, e\}$.
- **Step #3.4** Visit b.

Step #4 Since $|Q| \neq 0$, perform the next loop iteration for $v \in \{f, g\}$.

- **Step #4.1** Set $u = c$ and $Q = \langle d, e \rangle$.
- **Step #4.2** Since $f \notin D$, set $Q = \langle d, e, f \rangle$ and $D = \{a, b, c, d, e, f\}$.
- **Step #4.3** Since $g \notin D$, set $Q = \langle d, e, f, g \rangle$ and $D = \{a, b, c, d, e, f, g\}$.
- **Step #4.4** Visit c.

Step #5 Since $|Q| \neq 0$, perform the next loop iteration for $v \in \emptyset$.

[5] Why *search*? This terminology relates to how BFS and DFS are often used, namely to search for a target vertex within the graph: once the target vertex v is visited, the traversal usually stops rather than continuing to visit all vertices.

Step #5.1 Set $u = d$ and $Q = \langle e, f, g \rangle$.

Step #5.2 Visit c.

Step #6 Since $|Q| \neq 0$, perform the next loop iteration for $v \in \emptyset$.

Step #6.1 Set $u = e$ and $Q = \langle f, g \rangle$.

Step #6.2 Visit e.

Step #7 Since $|Q| \neq 0$, perform the next loop iteration for $v \in \emptyset$.

Step #7.1 Set $u = f$ and $Q = \langle g \rangle$.

Step #7.2 Visit f.

Step #8 Since $|Q| \neq 0$, perform the next loop iteration for $v \in \emptyset$.

Step #8.1 Set $u = g$ and $Q = \langle \rangle$.

Step #8.2 Visit g.

Step #9 Since $|Q| = 0$, stop the loop.

Step #10 Return.

Doing the same with the DFS algorithm produces a different behaviour:

Step #1 Set $Q = \langle a \rangle$ and $D = \{a\}$.

Step #2 Since $|Q| \neq 0$, perform the next loop iteration for $v \in \{b, c\}$.

Step #2.1 Set $u = a$ and $Q = \langle \rangle$.

Step #2.2 Since $b \notin D$, set $Q = \langle b \rangle$ and $D = \{a, b\}$.

Step #2.3 Since $c \notin D$, set $Q = \langle c, b \rangle$ and $D = \{a, b, c\}$.

Step #2.4 Visit a.

Step #3 Since $|Q| \neq 0$, perform the next loop iteration for $v \in \{f, g\}$.

Step #3.1 Set $u = c$ and $Q = \langle b \rangle$.

Step #3.2 Since $f \notin D$, set $Q = \langle f, b \rangle$ and $D = \{a, b, c, f\}$.

Step #3.3 Since $g \notin D$, set $Q = \langle g, f, b \rangle$ and $D = \{a, b, c, f, g\}$.

Step #3.4 Visit c.

Step #4 Since $|Q| \neq 0$, perform the next loop iteration for $v \in \emptyset$.

Step #4.1 Set $u = g$ and $Q = \langle f, b \rangle$.

Step #4.2 Visit c.

Step #5 Since $|Q| \neq 0$, perform the next loop iteration for $v \in \emptyset$.

Step #5.1 Set $u = f$ and $Q = \langle b \rangle$.

Step #5.2 Visit f.

Step #6 Since $|Q| \neq 0$, perform the next loop iteration for $v \in \{d, e\}$.

Step #6.1 Set $u = b$ and $Q = \langle \rangle$.

Step #6.2 Since $d \notin D$, set $Q = \langle d \rangle$ and $D = \{a, b, c, f, g, d\}$.

Step #6.3 Since $e \notin D$, set $Q = \langle e, d \rangle$ and $D = \{a, b, c, f, g, d, e\}$.

Step #6.4 Visit c.

Step #7 Since $|Q| \neq 0$, perform the next loop iteration for $v \in \emptyset$.

Step #7.1 Set $u = e$ and $Q = \langle d \rangle$.

Step #7.2 Visit e.

Step #8 Since $|Q| \neq 0$, perform the next loop iteration for $v \in \emptyset$.

Step #8.1 Set $u = d$ and $Q = \langle \rangle$.

Step #8.2 Visit d.

Step #9 Since $|Q| = 0$, stop the loop.

Step #10 Return.

To summarise, both BFS and DFS satisfy the goal of visiting all vertices but differ in the order they do so. You can think of this tree as three levels, where the top level contains a, the middle level contains b and c, and the bottom level contains d, e, f and g. BFS visits each vertex in the current level before moving onto a lower level; the order of traversal is

$$\langle a, b, c, d, e, f, g \rangle$$

which sort of moves across the tree then down. DFS on the other hand visits each vertex in the lower level before the current one; the order of traversal is therefore

$$\langle a, c, g, f, b, e, d \rangle,$$

which moves down the tree then across.

Using a tree is just a way to make the behaviours easier to explain however: we can of course invoke the same algorithms on a more general graph. If we take the web-graph in Figure 2 for example, the order BFS visits the web-pages is

⟨ `a.html`, `c.html`, `d.html`, `b.html`, `e.html`, `f.html` ⟩,

whereas for DFS this changes to

⟨ `a.html`, `d.html`, `f.html`, `e.html`, `b.html`, `c.html` ⟩.

Both BFS or DFS traverse the web-graph, so provided they act in the right way when visiting each web-page they (*more or less*) solve the original problem of automatically browsing the web.

In BFS and DFS, the way v is added to the worklist Q in line #7 is the central difference; formally, Q is used as a **queue** in BFS and as a **stack** in DFS. Find out more about these data structures.

By using a so-called `robots.txt` file [29], a web-server can signal that certain web-pages should be ignored by a web-crawler. Find out about this mechanism: when and why do you think it makes sense to use it?

2.2 Graph exploration

"*More or less* solves" is a hint: a subtle problem exists with using standard graph traversal. Remember that the web-crawler does not have access to the web-graph as input. The whole point is to automatically browse the web and collect information about web-pages, so if we already have the web-graph there would be no point!

Really solving this problem means translating BFS or DFS into **graph exploration** algorithms, but fortunately the difference is minor. Obviously we cannot have G as an input to the algorithm any longer; in a sense, the web-graph is now an output from exploration, not the input to traversal. As a result, line #5 cannot check for edges $(u, v) \in E$ because E is unknown. Instead, we need to take u and discover edges from it to other, potentially unknown web-pages. The form of web-pages makes this easy: HTML is a **mark-up language** [19], meaning as well as the actual content we see, each web-page contains extra information (the so-called mark-up). This mark-up will includes specifications of which web-page a given link is to, so if we parse the web-page content for `a.html` in Figure 2 for example, we might extract the edges (`a.html`, `c.html`) and (`a.html`, `d.html`) from the links `<a href='c.html'>link to c</a>` and `<a href='d.html'>link to d</a>` which are embedded in it.

So the idea is that we invoke the web-crawler on a starting point, for example a Top Level Domain (TLD) [33] such as s = `www.bbc.co.uk`, and let it explore links from there to construct G. Once finished (e.g., when it runs out of web-pages to explore) we might invoke it on *another* starting point and merge together the results (e.g., to increase the chance of visiting every web-page), or just use G as is for whatever purpose we had in mind.

In line #11 of Algorithm 1a and Algorithm 1b, we need to visit and process a web-page u. What this actually means depends on the task at hand of course: for some web-search engines this could just mean extracting and summarising the content of u, noting that `a.html` contains the words "apple" and "orange" for instance.

Since PageRank needs the actual web-graph structure (i.e., the links between web-pages) as input, the processing basically needs to form and eventually return $G = (V, E)$ by collecting the vertices and edges. Write an graph exploration algorithm to do this, using either Algorithm 1a or Algorithm 1b as a basis; demonstrate how it works using the web-graph in Figure 2.

3 Using probability theory to model web-browsing

Section 2 showed a suitable web-crawler can automatically generate a web-graph $G = (V, E)$ for us, where the edges in E capture the link structure required to reason about each web-page $x \in V$. The example in Figure 2 is quite detailed however: once we have G, it is easier to consider Figure 4a instead. This is the same graph (e.g., `a.html` is just renamed a) but any unnecessary detail such as the web-page content is removed.

Given a G as input, our next challenge is computation of a PageRank value for each web-page. So where do we start? As stated, this challenge is enormously vague: we lack a formal definition of *what* we should compute, *how* we should try to compute it, or even what the correct answer looks like! To make reasoning about the problem easier, Google use the following model: imagine the web is comprised of n web-pages, which a user browses indefinitely. At each step, this user randomly causes one of two events to occur:

1. with probability $1 - p$, a random web-page is loaded from anywhere on the web (i.e., it thinks of a random web-page and types the URL into the web-browser address bar), or
2. with probability p, it clicks on and hence follows a random link on the current web-page to some other web-page.

This is called the **random surfer model**, and amounts to performing a **random walk** [28] on the web-graph. Although this might not be how people *really* behave, it allows us to translate the vague English description into something more concrete: assuming we accept the random surfer model, the PageRank of a web-page is equivalent to the probability of visiting it. Think about it: if web-page x is more important then there will be more links to it, and as a result the probability is higher that at some point the random surfer visits x by following such a link (plus the probability it visits it at random of course). We can stress this by keeping in mind

$$\text{"the PageRank of } x\text{"} \quad \equiv \quad \Pr[x]$$

where the right-hand side means the probability that web-page x is visited; this in turn is equivalent to the number of votes accumulated by x if you prefer the voting analogy. It also allows a sanity check later when we come to compute the actual values: since we are dealing with probabilities, the sum of $\Pr[x]$ for all $x \in V$ should be 1.

This all becomes more concrete still by capturing the description as a formula:

$$\Pr[x] \quad = \quad \underbrace{\frac{1-p}{n}}_{\substack{\text{term representing} \\ \text{random web-page}}} \quad + \quad \underbrace{p \cdot \left(\sum_{y \in \text{into}(x)} \frac{\Pr[y]}{|\text{from}(y)|} \right)}_{\substack{\text{term representing} \\ \text{random link}}}$$

There are two terms because there are two events possible within the random surfer model; each term computes the associated probability, which we then add together because we want the probability of either one event or the other occurring as the result. Although they might *look* cryptic, both terms are just translations of the English description of random surfer behaviour into Mathematics. For the first term, this is easy to see: if there are n web-pages in total and we load a random one with probability $1 - p$ then we end up on x with a probability of $\frac{1-p}{n}$. The second is more difficult however. The term itself is p multiplied by

$$\sum_{y \in \text{into}(x)} \frac{\Pr[y]}{|\text{from}(y)|}$$

or, in English, the probability of this event occurring multiplied by the combined probabilities of arriving at x having followed a link from another web-page y. Think about it again using the voting analogy: a web-page

y has a number of votes to cast (i.e., $\Pr[y]$), so gives each each web-page x that it links to a number of votes in proportion to the total number of links it contains (i.e., divides $\Pr[y]$ by the number of outgoing links from y, namely $|\text{from}(y)|$). We are interested in x of course, so the summation basically deals with all web-pages y that link to x (i.e., all $y \in \text{into}(x)$), forming the sum of their votes cast for x per the above. This means two things:

1. if y has many votes to cast, the number given to x will be larger (since the numerator $\Pr[y]$ will be larger) than if it had few, and
2. if y votes for fewer web-pages by linking to them, the proportion given to x will be larger (since the denominator $|\text{from}(y)|$ will be smaller) than if it votes for many.

By replacing votes with probabilities we get the desired result: there is a higher probability the random surfer visits x by following a link if other web-pages link to it, and even more so if those web-pages have a high probability of being visited themselves.

Although PageRank is probably the most famous, other similar examples exist within a family of related techniques. Two such examples are

- the **impact factor** [16] used to gauge how important an academic publication is, and
- the **Hyperlink-Induced Topic Search (HITS)** algorithm [12] that deals with web-page ranking.

It is often important to see how techniques relate or build on each other: do some research into the above, and compare them with PageRank in terms of their approach, features and so on.

3.1 Sanitising the web-graph to avoid a subtle problems

Two special types of web-page can occur in a web-graph, namely

1. **source web-pages**, which have no incoming links (i.e., $\text{into}(x) = \emptyset$) so will never be *visited* by following links, and
2. **sink web-pages**, which have no outgoing links (i.e., $\text{from}(x) = \emptyset$) so will never be *exited* by following links.

For the first case, the only potential problem might be that our web-crawler fails to visit it. In terms of computing PageRank values, there is no issue: it is just deemed unimportant according to the PageRank metric, due to the lack of links to it. The second case *is* problematic however. There are two ways to think about why this is the case:

1. The random surfer model says one of two events will occur: either the random surfer loads a random web-page or follows a random link. If the random surfer visits a sink web-page however, the first event cannot occur because there are no links to follow. Intuitively this is a problem because it means the probabilistic model of behaviour we rely on breaks down for sink web-pages.
2. In the corresponding formula, each web-page can be seen as voting for others by distributing the votes allocated to it via the links it contains. A sink web-page votes for no other web-page however, even though they *might* vote for it. Numerically this is a problem because a sink web-page accumulates votes like a sort of black hole. That is, votes flow in but never comes out again; this skews the results produced for non-sink web-pages because there is a fixed number of votes in total.

As a result, it is common to treat sink nodes differently. Various ways to do this have been proposed, but the easiest to justify is as follows: for each sink web-page x identified, we add an artificial edge of the form (x, y) to to every other vertex $y \in V$. By doing so, the random surfer is happy again because either event can occur once it visits x; it also ensures fairness to non-sink web-pages, because the PageRank accumulated by x is now shared evenly among all other web-pages.

Consider Figure 4a: web-page f satisfies the criteria for being a sink, since $\text{from}(f) = \emptyset$. To cope, we might amend the web-graph to produce Figure 4b where the dashed edges have been added artificially; these ensure f is linked to every other web-page, in this case a, b, c, d and e. We use this altered web-graph rather than the original from here on.

An aside: root finding with the iterative Newton-Raphson method.

A good example to illustrate the concept of using iterative methods is the Newton-Raphson [21] method for computing the **roots** of a function δ, i.e., an x such that

$$\delta(x) = 0.$$

If you think about the case of finding the square root of an integer y, this amounts to finding an x such that $x^2 = y$ meaning $\delta(x) = x^2 - y = 0$. Skipping a lot of theory that explains why, we do so using the relationship

$$x_{i+1} = x_i - \frac{\delta(x_i)}{\delta'(x_i)}$$

where δ' is the derivative of δ; here, since $\delta(x) = x^2 - y = 0$, we know $\delta'(x) = 2 \cdot x$. The important thing is that the left-hand side shows how to compute the $(i+1)$-th element in a sequence of progressively more accurate solutions, given the i-th such element on the right-hand side. For example, imagine we have $y = 4321$ and want to compute $x = \sqrt{y}$. First we make a guess at x, say $x_0 = 100$, and then iterate use of the recurrence to produce the following sequence:

$$\begin{array}{rcrcrcrcrcr}
x_1 & = & x_0 & - & \frac{\delta(x_0)}{\delta'(x_0)} & = & 100.000 & - & \frac{5679.000}{200.000} & = & 71.605 \\
x_2 & = & x_1 & - & \frac{\delta(x_1)}{\delta'(x_1)} & = & 71.605 & - & \frac{806.276}{143.210} & = & 65.974 \\
x_3 & = & x_2 & - & \frac{\delta(x_2)}{\delta'(x_2)} & = & 65.974 & - & \frac{31.697}{131.949} & = & 65.734 \\
x_4 & = & x_3 & - & \frac{\delta(x_3)}{\delta'(x_3)} & = & 65.734 & - & \frac{0.057}{131.468} & = & 65.734 \\
x_5 & = & x_4 & - & \frac{\delta(x_4)}{\delta'(x_4)} & = & 65.734 & - & \frac{0.000}{131.468} & = & 65.734 \\
\vdots & & & & & & \vdots & & & & \vdots
\end{array}$$

Eventually, the changes in our solution get very small and we can say they **converge**. We need a **termination criteria** to detect this, but in our example this is easy since we can check whether $\text{abs}(x_i^2 - y)$ is small enough to consider that x_i as correct: here we might say that because

$$\text{abs}(65.734^2 - 4321) = \text{abs}(4320.958 - 4321) = 0.041$$

is small enough, $x = x_5 = 65.734 \simeq \sqrt{4321}$ is an acceptable solution.

The "link sink web-page x to all other web-pages" strategy described is not the *only* option: various other strategies have also been proposed. Find out about at least one other strategy, then compare and contrast each option using a list of advantages and disadvantages.

Write an algorithm that takes a web-graph as input, and applies a strategy (whether the one described, or discovered in Task 5) for dealing with sink web-pages; the output should be the amended web-graph ready for use during computation of PageRank values.

If you consider the *real* web rather than the examples presented, the value of n is large and hence many web-pages will be identified as sinks. If you implement the strategy described for dealing with them *exactly*, a problem starts to emerge. What do you think this problem could be, and, with reference to your algorithm in Task 6 for instance, how could it be avoided?

3.2 A Mathematical approach to computing PageRank

3.2.1 Rewriting the formula to avoid cycles

Now armed with a web-graph *and* a formula to compute $\Pr[x]$ for each web-page, you could be forgiven for thinking that we are done. There is a subtle but important problem lucking behind the scenes however. Look again at

$$\Pr[x] = \frac{1-p}{n} + p \cdot \left(\sum_{y \in \text{into}(x)} \frac{\Pr[y]}{|\text{from}(y)|} \right),$$

keeping in mind the goal is to compute the left-hand side. How? There is no hidden trick: to get the left-hand side, we just evaluate the right-hand side as with any equality. But the right-hand side includes $\Pr[y]$, so how do we compute this? Use the formula again! Now we have a chicken-and-egg style problem: we cannot compute $\Pr[x]$ until we already know $\Pr[y]$ for all $y \in V$, which is then a cyclic argument (*literally*, because the problem stems from cycles in the web-graph).

Fortunately, we can resolve this using some slightly more advanced probability theory. The basic idea is that the random surfer model implicitly includes a notion of time: the random surfer browses web-pages indefinitely, but it does so step-by-step. Instead of thinking of $\Pr[x]$, as we have done so far, we make an alteration by letting $\Pr[x^{(t)}]$ denote the probability that the random surfer visits web-page x in step (or at time) t. If we count the steps as starting at $t = 0$, then

$$\Pr[a^{(0)}] = \Pr[b^{(0)}] = \Pr[c^{(0)}] = \Pr[d^{(0)}] = \Pr[e^{(0)}] = \Pr[f^{(0)}] = \frac{1}{6}.$$

Hopefully this makes sense: we have to start *somewhere*, and since there are $n = 6$ web-pages then each one has probability $\frac{1}{n} = \frac{1}{6}$ of being the starting point. What now? Well, at step $t = 1$ one or other of the events occurs and the random surfer visits another web-page. Imagine the random surfer starts by visiting web-page b at step $t = 0$ for example; what is the probability it visits web-page d at step $t + 1$? We can answer this by looking at how probable the two events are in this case:

1. It might visit d with probability $1 - p$, by loading the web-page at random. There is a $\frac{1}{n}$ probability of visiting any specific web-page of the n in total, so a $\frac{1}{6}$ probability of visiting d. This means a $\frac{1-p}{6}$ probability overall.

2. It might visit d with probability p, by following a random link. There are $|\text{from}(y)|$ links from any given web-page y, so $|\text{from}(b)| = |\{a, d, e\}| = 3$ from b specifically. Only one of those links is to d though, so the probability of following that one specifically is $\frac{1}{3}$. This means a $\frac{p}{3}$ probability overall.

Putting this together, we can write

$$\Pr[d^{(t+1)} \mid b^{(t)}] = \frac{1-p}{6} + \frac{p}{3}$$

where the left-hand side means the probability of visiting web-page d in step $t + 1$ having visited web-page b in step t; this is an example of **conditional probability** [4].

Of course this is only one way we might visit d. To be complete, we need to take into account that we could follow a link from *any* web-page to it. We can follow the same reasoning as above, and find the following:

$$\begin{aligned}
\Pr[d^{(t+1)} \mid a^{(t)}] &= \tfrac{1-p}{6} + \tfrac{p}{2} \\
\Pr[d^{(t+1)} \mid b^{(t)}] &= \tfrac{1-p}{6} + \tfrac{p}{3} \\
\Pr[d^{(t+1)} \mid c^{(t)}] &= \tfrac{1-p}{6} + \tfrac{p}{1} \\
\Pr[d^{(t+1)} \mid d^{(t)}] &= \tfrac{1-p}{6} + 0 \\
\Pr[d^{(t+1)} \mid e^{(t)}] &= \tfrac{1-p}{6} + 0 \\
\Pr[d^{(t+1)} \mid f^{(t)}] &= \tfrac{1-p}{6} + \tfrac{p}{5}
\end{aligned}$$

What we want though, is what is the probability of visiting web-page d in step $t + 1$ outright *not* as part of some condition. All this means is we combine all possible ways of visiting d per the above. Keeping in mind that

the sum of $\Pr[y^{(t)}]$ for all $y \in V$ must be 1 since these are probabilities, we end up with the following:

$$\begin{aligned}
\Pr[d^{(t+1)}] &= \Pr[d^{(t+1)} \mid a^{(t)}] \cdot \Pr[a^{(t)}] + \Pr[d^{(t+1)} \mid b^{(t)}] \cdot \Pr[b^{(t)}] + \\
&\quad \Pr[d^{(t+1)} \mid c^{(t)}] \cdot \Pr[c^{(t)}] + \Pr[d^{(t+1)} \mid d^{(t)}] \cdot \Pr[d^{(t)}] + \\
&\quad \Pr[d^{(t+1)} \mid e^{(t)}] \cdot \Pr[e^{(t)}] + \Pr[d^{(t+1)} \mid f^{(t)}] \cdot \Pr[f^{(t)}] \\
&= (\tfrac{1-p}{6} + \tfrac{p}{2}) \cdot \Pr[a^{(t)}] + (\tfrac{1-p}{6} + \tfrac{p}{3}) \cdot \Pr[b^{(t)}] + \\
&\quad (\tfrac{1-p}{6} + \tfrac{p}{1}) \cdot \Pr[c^{(t)}] + (\tfrac{1-p}{6} + 0) \cdot \Pr[d^{(t)}] + \\
&\quad (\tfrac{1-p}{6} + 0) \cdot \Pr[e^{(t)}] + (\tfrac{1-p}{6} + \tfrac{p}{5}) \cdot \Pr[f^{(t)}] \\
&= \tfrac{1-p}{6} + p \cdot \left(\frac{\Pr[a^{(t)}]}{2} + \frac{\Pr[b^{(t)}]}{3} + \frac{\Pr[c^{(t)}]}{1} + \frac{\Pr[f^{(t)}]}{5} \right)
\end{aligned}$$

Each term on the right-hand side multiplies the probability of visiting some web-page x in step $t+1$ having visited another one y at step t, i.e.,

$$\Pr[x^{(t+1)} \mid y^{(t)}],$$

with the probability of actually having visited y at step t, i.e.,

$$\Pr[y^{(t)}].$$

All such terms are added together, because we want to know the probability of any one of them occurring. Doing a similar thing for each web-page, we end up with

$$\begin{aligned}
\Pr[a^{(t+1)}] &= \tfrac{1-p}{6} + p \cdot \left(\frac{\Pr[b^{(t)}]}{3} + \frac{\Pr[f^{(t)}]}{5} \right) \\
\Pr[b^{(t+1)}] &= \tfrac{1-p}{6} + p \cdot \left(\frac{\Pr[d^{(t)}]}{3} + \frac{\Pr[f^{(t)}]}{5} \right) \\
\Pr[c^{(t+1)}] &= \tfrac{1-p}{6} + p \cdot \left(\frac{\Pr[a^{(t)}]}{2} + \frac{\Pr[f^{(t)}]}{5} \right) \\
\Pr[d^{(t+1)}] &= \tfrac{1-p}{6} + p \cdot \left(\frac{\Pr[a^{(t)}]}{2} + \frac{\Pr[b^{(t)}]}{3} + \frac{\Pr[c^{(t)}]}{1} + \frac{\Pr[f^{(t)}]}{5} \right) \\
\Pr[e^{(t+1)}] &= \tfrac{1-p}{6} + p \cdot \left(\frac{\Pr[b^{(t)}]}{3} + \frac{\Pr[d^{(t)}]}{3} + \frac{\Pr[f^{(t)}]}{5} \right) \\
\Pr[f^{(t+1)}] &= \tfrac{1-p}{6} + p \cdot \left(\frac{\Pr[d^{(t)}]}{3} + \frac{\Pr[e^{(t)}]}{1} \right)
\end{aligned}$$

or, generalising this for *any* given web-page x,

$$\Pr[x^{(t+1)}] = \frac{1-p}{n} + p \cdot \left(\sum_{y \in \text{into}(x)} \frac{\Pr[y^{(t)}]}{|\text{from}(y)|} \right).$$

Unsurprisingly, this looks a like the formula we started with. The crucial difference is that on the left-hand side we now refer to step $t+1$ (values relating to which are unknown), whereas on the right-hand side we *only* refer to step t (values relating to which we know). That might not *seem* like a big difference, but is solves our chicken-and-egg problem: finally, we can compute actual PageRank values.

3.2.2 Using an iterative method to compute results

To do so, we draw on use of **iterative methods** [18] elsewhere in Mathematics. The underlying idea is that instead of computing a solution directly, we compute a sequence of approximate solutions each of which is more accurate (i.e., closer to the real solution) than the last: we start with an approximation x_0, then use a function δ to iteratively compute

$$x_{i+1} = \delta(x_i)$$

forming the next, $(i+1)$-th approximation from the last, i-th one. At some i-th step the process will converge, meaning x_i is then accurate enough to accept as the solution. This general description should seem similar to what we developed above. Remember that we want to compute

$$\Pr[x^{(t+1)}] = \frac{1-p}{n} + p \cdot \left(\sum_{y \in \text{into}(x)} \frac{\Pr[y^{(t)}]}{|\text{from}(y)|} \right),$$

so, following the above, δ is basically just the PageRank formula with a right-hand side relating to the last, t-th approximation and a left-hand side relating to the next, $(t+1)$-th approximation. To adopt the same approach

however, we need to resolve one minor difference: in the general description we are only interested in one x, whereas we want a solution for each web-pages in the web-graph, i.e., all $x \in V$ or n solutions in total. To cope, we combine all n formula into one using the matrix form of what is then a system of n **linear equations** [31]:

$$\begin{pmatrix} \Pr[a^{(t+1)}] \\ \Pr[b^{(t+1)}] \\ \Pr[c^{(t+1)}] \\ \Pr[d^{(t+1)}] \\ \Pr[e^{(t+1)}] \\ \Pr[f^{(t+1)}] \end{pmatrix} = \left(\frac{1-p}{n}\right) \cdot \begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{pmatrix} + p \cdot \begin{pmatrix} 0 & \frac{1}{3} & 0 & 0 & 0 & \frac{1}{5} \\ 0 & 0 & 0 & \frac{1}{3} & 0 & \frac{1}{5} \\ \frac{1}{2} & 0 & 0 & 0 & 0 & \frac{1}{5} \\ \frac{1}{2} & \frac{1}{3} & 1 & 0 & 0 & \frac{1}{5} \\ 0 & \frac{1}{3} & 0 & \frac{1}{3} & 0 & \frac{1}{5} \\ 0 & 0 & 0 & \frac{1}{3} & 1 & 0 \end{pmatrix} \cdot \begin{pmatrix} \Pr[a^{(t)}] \\ \Pr[b^{(t)}] \\ \Pr[c^{(t)}] \\ \Pr[d^{(t)}] \\ \Pr[e^{(t)}] \\ \Pr[f^{(t)}] \end{pmatrix}$$

This could be simplified into

$$\vec{x}^{(t+1)} = \delta(\vec{x}^{(t)}) = \vec{b} + A \cdot \vec{x}^{(t)}$$

where $\vec{x}$ is now a column vector capturing the PageRank values for *all* web-pages rather than just one, while A and $\vec{b}$ are a constant column vector and a matrix respectively (the latter of which is derived from link structure in the web-graph). We already informally decided that an appropriate initial approximation would be

$$\vec{x}^{(0)} = \begin{pmatrix} \frac{1}{n} \\ \frac{1}{n} \\ \vdots \\ \frac{1}{n} \end{pmatrix}$$

i.e., an n-element column vector whose elements are all $\frac{1}{n}$. So, using this we can proceed to iteratively compute

$$\begin{array}{rcll} \vec{x}^{(1)} & = & \delta(\vec{x}^{(0)}) & = \vec{b} + A \cdot \vec{x}^{(0)} \\ \vec{x}^{(2)} & = & \delta(\vec{x}^{(1)}) & = \vec{b} + A \cdot \vec{x}^{(1)} \\ & & & = \vec{b} + A \cdot \left(\vec{b} + A \cdot \vec{x}^{(0)}\right) \\ & & & = \vec{b} + A \cdot \vec{b} + A^2 \vec{x}^{(0)} \\ & \vdots & & \vdots \\ \vec{x}^{(t+1)} & = & \delta(\vec{x}^{(t)}) & = \left(\sum_{i=0}^{t-1} A^i\right) \cdot \vec{b} + A^t \cdot \vec{x}^{(0)} \end{array}$$

until at some t-th step the process converges, meaning $\vec{x}^{(t)}$ is then accurate enough to accept as the solution.

If you consider the *real* web rather than the examples presented, the value of n is large and hence the value of $\Pr[x]$ for any given x will be very small; numerical precision [26] becomes a problem. How can this problem be resolved?

3.2.3 Selecting the probability p

The only remaining question is what value we should choose for p, the probability which controls the random surfer model: a larger p means there is a higher probability the random surfer opts to follow a random link, whereas a smaller p means a higher probability it loads a random web-page. It is quite important to find a balance, because taken to an extreme,

- if p is *too* large then the random surfer might never encounter poorly connected web-pages at all (since it is less likely to load them at random), whereas
- if p is *too* small then we sort of ignore the web-graph structure (since it is less likely to follow any given link), which of course contradicts the original aim.

A little more formally, p also influences how quickly the iterative method will arrive at a solution: a larger p places more emphasis on following links, meaning their influence will spread[6] more quickly, and vice versa. In their original research paper describing a prototype Google web-search system, Sergey Brin and Larry Page quote $p = 0.85$; it is less clear what Google use now, but we will stick with this as a reasonable guess.

[6] This also explains why in various descriptions, including the original research paper, p is termed a **damping factor** (or **damping ratio**); this term stems from description of a similar feature of physical systems [6].

3.2.4 Concrete PageRank values for the example web-graph

Now, *finally*, we can actually compute the PageRank values themselves. Recall that we now have a formula

$$\vec{x}^{(t+1)} = \delta(\vec{x}^{(t)}) = \vec{b} + A \cdot \vec{x}^{(t)}$$

which allows an iterative method of computing a solution; each part of the formula is now specified, in the sense that our example web-graph shown in Figure 4b tells us that $n = 6$ and

$$A = \begin{pmatrix} 0 & \frac{1}{3} & 0 & 0 & 0 & \frac{1}{5} \\ 0 & 0 & 0 & \frac{1}{3} & 0 & \frac{1}{5} \\ \frac{1}{2} & 0 & 0 & 0 & 0 & \frac{1}{5} \\ \frac{1}{2} & \frac{1}{3} & 1 & 0 & 0 & \frac{1}{5} \\ 0 & \frac{1}{3} & 0 & \frac{1}{3} & 0 & \frac{1}{5} \\ 0 & 0 & 0 & \frac{1}{3} & 1 & 0 \end{pmatrix}.$$

Likewise, we know that

$$\vec{b} = \begin{pmatrix} \frac{1}{40} \\ \frac{1}{40} \\ \frac{1}{40} \\ \frac{1}{40} \\ \frac{1}{40} \\ \frac{1}{40} \end{pmatrix}$$

and

$$\vec{x}^{(0)} = \begin{pmatrix} \frac{1}{6} \\ \frac{1}{6} \\ \vdots \\ \frac{1}{6} \end{pmatrix}.$$

We therefore produce the following sequence of approximate but concrete solutions:

	$\vec{x}^{(0)}$	$\vec{x}^{(1)}$	$\vec{x}^{(2)}$	$\vec{x}^{(3)}$	$\vec{x}^{(4)}$	$\vec{x}^{(5)}$	$\cdots$
$\Pr[a^{(t)}]$	0.166	0.101	0.090	0.108	0.104	0.104	$\cdots$
$\Pr[b^{(t)}]$	0.166	0.101	0.150	0.133	0.133	0.134	$\cdots$
$\Pr[c^{(t)}]$	0.166	0.124	0.104	0.104	0.113	0.110	$\cdots$
$\Pr[d^{(t)}]$	0.166	0.313	0.238	0.235	0.239	0.244	$\cdots$
$\Pr[e^{(t)}]$	0.166	0.148	0.179	0.176	0.171	0.171	$\cdots$
$\Pr[f^{(t)}]$	0.166	0.214	0.239	0.244	0.241	0.238	$\cdots$

We could continue of course, but the values in iteration 4 versus those in iteration 5 already show little change. Using this as a termination criteria, we take $\vec{x}^{(5)}$ as our solution: these are the probabilities the random surfer will visit each web-page in our web-graph. For instance, it visits a with probability 0.104, i.e., about 10 percent of the time.

The results presented above relate to Figure 4b, i.e., the web-graph that has been amended to cope with any sink web-pages (in this case just f). Try to

1. reproduce these results yourself, then
2. do the same thing with Figure 4a, the original web-graph.

Compare the results with each other: what do you notice, and how do you explain each identifiable difference?

4 Putting it all together: using PageRank to produce web-search results

As noted, PageRank is only one component in the Google web-search system: it forms part of the final stage outlined in Section 1.2.2 by ranking (or sorting) the set of results produced for some search query. To wrap-up and meet the challenge of explaining how the system works, it makes sense to look at how and where PageRank fits in. To start with, in the offline phase, we

1. use a web-crawler to build a web-graph and summary of content for each web-page, then

2. compute PageRank values for each web-page in the web-graph.

For our limited example, the outcome of this phase can be summarised as follows

$$\begin{array}{rl} \{\text{“apple”}, \text{“orange”}\} \in a & \Pr[a] = 0.104 \\ \{\text{“apple”}, \text{“banana”}\} \in b & \Pr[b] = 0.134 \\ \{\text{“banana”}, \text{“orange”}\} \in c & \Pr[c] = 0.110 \\ \{\text{“apple”}, \text{“banana”}\} \in d & \Pr[d] = 0.244 \\ \{\text{“apple”}, \text{“pear”}\} \in e & \Pr[e] = 0.171 \\ \{\text{“banana”}, \text{“pear”}\} \in f & \Pr[f] = 0.238 \end{array}$$

in the sense that web-page a is viewed as containing content relating to the words “apple” and “orange”, and has a PageRank of 0.104. Now, imagine a user gives us a search query q during the online phase. We need to

1. match web-pages with the query in order to build R, an initial set of results, then
2. take the web-pages in R, and sort them according to their PageRank values to produce the results actually presented for the user.

The first step depends a lot on the type of queries we want to allow, but given the simple form of content our web-pages house, imagine a simple form of query that is just a single word: we want the set of results to give us the web-pages which are most relevant, ranked by their PageRank-decided importance. Consider the following for example:

- If $q =$ “orange”, the initial set of results is $R = \{a, c\}$. Given

$$\begin{array}{rcl} \Pr[a] & = & 0.104 \\ \Pr[c] & = & 0.110 \end{array}$$

  the results are displayed with web-page c first, then a.

- If $q =$ “apple”, the initial set of results is $R = \{a, b, d, e\}$. Given

$$\begin{array}{rcl} \Pr[a] & = & 0.104 \\ \Pr[b] & = & 0.134 \\ \Pr[d] & = & 0.244 \\ \Pr[e] & = & 0.171 \end{array}$$

  the results are displayed with web-page d first, then e, b and finally a.

Even with such simple web-pages and queries, you see the PageRank concept in action. For instance, a, b, d and e are all relevant for the query “apple”. Both a and b have links to d: each link source can be viewed as attesting to the importance of the link target, meaning d ends up with a (relatively) high PageRank value and is displayed first.

Search Engine Optimisation (SEO) [30] is the art of designing web-pages so they appear in web-search results more often and/or with a higher ranking than normal; this might be used in a marketing strategy, where the overall goal is that users visit the web-page more often.

Various acceptable and unacceptable SEO methods exist: do some research into both sides of this arms race, i.e.,

1. how web-page owners might try to inflate their PageRank and hence get ranked higher in a set of results, and
2. how Google identify and eliminate unfair SEO practices which skew their results and, arguably, devalue PageRank.

Given a small example, it is hard to see the value PageRank gives in general. There are two related take-away points from this Chapter. First, once n grows large enough, a huge number of web-pages will always be deemed relevant for any given query. Put another way, without PageRank we would almost be back to square one: there would be so many web-pages like a, b and e that match, without PageRank to help us we would be tasked with picking out d by hand. Second, given the benefit PageRank provides, you may have previously thought of it as complex or even magic in some way (if you knew it existed at all). In reality however, we have explained more or less the entire thing by taking a 2-step strategy:

1. model various problems in a way we can more easily understand and reason about, and
2. apply fundamental but fairly simple Mathematics (i.e., probability and graph theory, and iterative methods) and Computer Science (i.e., algorithms) to produce solutions.

Although the techniques themselves might not be applicable to all problems, the general strategy *is*. This makes PageRank a great example, and advert, for Computer Science as a whole.

References

[1] *Wikipedia: AdSense*. https://en.wikipedia.org/wiki/AdSense (see p. 3).

[2] *Wikipedia: AdWords*. http://en.wikipedia.org/wiki/AdWords (see p. 3).

[3] *Wikipedia: Breadth-first search*. https://en.wikipedia.org/wiki/Breadth-first_search (see p. 10).

[4] *Wikipedia: Conditional probability*. http://en.wikipedia.org/wiki/Conditional_probability (see p. 16).

[5] *Wikipedia: Cycle*. http://en.wikipedia.org/wiki/Cycle_(graph_theory) (see p. 10).

[6] *Wikipedia: Damping ratio*. http://en.wikipedia.org/wiki/Damping_ratio (see p. 18).

[7] *Wikipedia: Depth-first search*. http://en.wikipedia.org/wiki/Depth-first_search (see p. 10).

[8] *Wikipedia: Graph*. http://en.wikipedia.org/wiki/Graph_(mathematics) (see p. 9).

[9] *Wikipedia: Graph theory*. http://en.wikipedia.org/wiki/Graph_theory (see p. 9).

[10] *Wikipedia: Graph traversal*. http://en.wikipedia.org/wiki/Graph_traversal (see p. 6).

[11] *Wikipedia: Hyperlink*. http://en.wikipedia.org/wiki/Hyperlink (see p. 4).

[12] *Wikipedia: Hyperlink-Induced Topic Search (HITS)*. http://en.wikipedia.org/wiki/HITS_algorithm (see p. 14).

[13] *Wikipedia: Hypertext*. http://en.wikipedia.org/wiki/Hypertext (see p. 4).

[14] *Wikipedia: HyperText Mark-up Language (HTML)*. http://en.wikipedia.org/wiki/HTML (see p. 4).

[15] *Wikipedia: HyperText Transfer Protocol (HTTP)*. http://en.wikipedia.org/wiki/Hypertext_Transfer_Protocol (see p. 4).

[16] *Wikipedia: Impact factor*. http://en.wikipedia.org/wiki/Impact_factor (see p. 14).

[17] *Wikipedia: Information retrieval*. http://en.wikipedia.org/wiki/Information_retrieval (see p. 4).

[18] *Wikipedia: Iterative method*. http://en.wikipedia.org/wiki/Iterative_method (see p. 17).

[19] *Wikipedia: Mark-up language*. http://en.wikipedia.org/wiki/Markup_language (see p. 12).

[20] *Wikipedia: Meta element*. http://en.wikipedia.org/wiki/Meta_element (see p. 5).

[21] *Wikipedia: Newton-Raphson method*. https://en.wikipedia.org/wiki/Newton's_method (see p. 15).

[22] *Wikipedia: Online and offline*. http://en.wikipedia.org/wiki/Online_and_offline (see p. 5).

[23] *Wikipedia: oN-Line System (NLS)*. http://en.wikipedia.org/wiki/NLS_(computer_system) (see p. 4).

[24] *Wikipedia: Open Directory Project (ODP)*. http://en.wikipedia.org/wiki/Open_Directory_Project (see p. 4).

[25] *Wikipedia: PageRank*. http://en.wikipedia.org/wiki/PageRank (see p. 5).

[26] *Wikipedia: Precision*. http://en.wikipedia.org/wiki/Precision_(computer_science) (see p. 18).

[27] *Wikipedia: Pre-computation*. http://en.wikipedia.org/wiki/Precomputation (see p. 5).

[28] *Wikipedia: Random walk*. http://en.wikipedia.org/wiki/Random_walk (see p. 13).

[29] *Wikipedia: Robots exclusion standard*. http://en.wikipedia.org/wiki/Robots_Exclusion_Standard (see p. 12).

[30] *Wikipedia: Search engine optimization*. http://en.wikipedia.org/wiki/Search_engine_optimization (see p. 20).

[31] *Wikipedia: System of linear equations*. https://en.wikipedia.org/wiki/System_of_linear_equations (see p. 18).

[32] *Wikipedia: The mother of all demos*. http://en.wikipedia.org/wiki/The_Mother_of_All_Demos (see p. 4).

[33] *Wikipedia: Top Level Domain (TLD)*. http://en.wikipedia.org/wiki/Top-level_domain (see p. 12).

[34] *Wikipedia: Tree*. https://en.wikipedia.org/wiki/Tree_(graph_theory) (see p. 10).

[35] *Wikipedia: Web indexing*. http://en.wikipedia.org/wiki/Web_indexing (see p. 5).

[36] *Wikipedia: Web-crawler*. http://en.wikipedia.org/wiki/Web_crawler (see p. 5).

[37] *Wikipedia: Web-directory*. http://en.wikipedia.org/wiki/Web_directory (see p. 4).

[38] *Wikipedia: Web-graph*. http://en.wikipedia.org/wiki/Webgraph (see p. 6).

[39] *Wikipedia: Web-search engine*. http://en.wikipedia.org/wiki/Web_search_engine (see p. 3).

[40] *Wikipedia: Web-search query*. http://en.wikipedia.org/wiki/Web_search_query (see p. 4).

[41] *Wikipedia: World Wide Web*. http://en.wikipedia.org/wiki/World_Wide_Web (see p. 3).

[42] *Wikipedia: World Wide Web Virtual Library (WWWVL)*. http://en.wikipedia.org/wiki/World_Wide_Web_Virtual_Library (see p. 4).

I have a question/comment/complaint for you. Any (positive *or* negative) feedback, experience or comment is very welcome; this helps us to improve and extend the material in the most useful way.

However, keep in mind we are far from perfect; mistakes are of course possible, although hopefully rare. Some cases are hard for us to check, and make your feedback even more valuable: for instance

1. minor variation in software versions can produce subtle differences in how some commands and hence examples work, and
2. some examples download and use online resources, but web-sites change over time (or even might differ depending on where you access them from) so might cause the example to fail.

Either way, if you spot a problem then let us know: we will try to explain and/or fix things as fast as we can!

Can I use this material for something ? We are distributing these PDFs under the Creative Commons Attribution-ShareAlike License (CC BY-SA)

http://creativecommons.org/licenses/by-sa/4.0/

This is a fancy way to say you can basically do whatever you want with them (i.e., use, copy and distribute it however you see fit) as long as a) you correctly attribute the original source, and b) you distribute the result under the same license.

Is there a printed version of this material I can buy? Yes: Springer have published selected Chapters in

http://www.springer.com/computer/book/978-3-319-04041-7

while *also* allowing us to keep electronic versions online. Any royalties from this published version are donated to the UK-based Computing At School (CAS) group, whose ongoing work can be followed at

http://www.computingatschool.org.uk/

Why are all your references to Wikipedia? Our goal is to give an easily accessible overview, so it made no sense to reference lots of research papers. There are basically two reasons why: research papers are often written in a way that makes them hard to read (even when their intellectual content is not difficult to understand), and although many research papers are available on the Internet, many are not (or have to be paid for). So although some valid criticisms of Wikipedia exist, for introductory material on Computer Science it certainly represents a good place to start.

I like programming; why do the examples include so little programming? We want to focus on interesting topics rather than the mechanics of programming. So even when we include example programs, the idea is to do so in a way where their meaning is fairly clear. For example it makes more sense to use pseudo-code algorithms or reuse existing software tools than complicate a description of something by including pages and pages of program listings.

But you need to be able to program to do Computer Science, right? Yes! But only in the same way as you need to be able to read and write to study English. Put another way, reading and writing, or grammar and vocabulary, are just tools: they simply allow us to study topics such as English literature. Computer Science is the same. Although it *is* possible to study programming as a topic in itself, we are more interested in what can be achieved *using* programs: we treat programming itself as another tool.