Com S 435/535: Notes VIII

1 Web Graph

The world wide web can be modeled as a directed graph $G = (V, E)$, where V is the set of all web pages. There is a directed edge from page p to page q if p has an *outgoing link* to q. Thus the out degree of a page p is the number of (unique) links that appear in that page. The in degree of a page p is the number of (unique) pages that contain a link to page p.

One of the first steps in building a search engine is to build a *web crawler*— A program that collects contents of all web pages. Depending on application, one may choose to write a crawler that collects contents of all (possible) web pages; or one may choose to write a crawler that collects web pages pertaining to a specific topic (such as sports, politics etc).

A web crawlers collects the data from web pages by traversing the web graph. While there are many known algorithms that can be used to traverse a graph, Breadth First Search (or its variants) seems to be the best choice to traverse the web graph. Let us recall the BFS algorithm. Given a directed graph $G = (V, E)$ and a node called *root* in V as input the algorithm works as follows. The algorithm maintains two lists—Queue denoted as Q and a list of vertices called *visited*. Initially both Q and *visited* contain only one vertex, the *root* vertex. The algorithm repeatedly performs the following: Look at the first node v from Q. For every outgoing edge $\langle v, u\rangle$ check if u is in *visited*; if not add u to the end of Q and *visited*. Below is the pseudo code.

```
1. Input Directed Graph $G = (V, E)$, and $root \in V$.
2. Initialize a Queue $Q$ and a list visited.
3. Place root in $Q$ and visited.
4. while $Q$ is not empty Do
   (a) Let $v$ be the first element of $Q$.
   (b) For every edge $\langle v, u\rangle \in E$ DO
       • If $u \notin visited$ add $u$ to (at end of $Q$) $Q$ and visited.
```

This procedure will visit all the vertices that are reachable from *root*.

Using BFS, we can build a web crawler. The main difference is that in the above description of BFS we assumed that the entire graph G is given as input and can be stored in memory; however that is not the case with the Web Graph. We explore the graph using BFS. We start with a root commonly known as *seed url*.

1. Input: *seed url*
2. Initialize a Queue Q and a list *visited*.
3. Place *seed url* in Q and *visited*.
4. **while** Q is not empty **Do**
   (a) Let $currentPage$ be the first element of Q.
   (b) Send a request to server at $currentPage$ and download $currentPage$.
   (c) Extract all links from $currentPage$.
   (d) For every link u that appears in $currentPage$
      - If $u \notin visited$ add u to Q and *visited*.

This crawler will download all web pages that can be reached from the *seed url*.

1.1 Practical Issues in Crawling

Though crawling is conceptually simpler (as it is simply BFS), implementing a crawler is somewhat challenging. Clearly, we want the crawler to run as fast as possible. The most time consuming part of the crawling is that of *sending a request to server and wait for its response.* During this time the crawler is idle. To reduce this inefficiency, typical cralwers use multiple threads and each thread sends requests to servers; thus we can fetch a few hundred pages at once. If our crawler program uses t threads, then we retrieve first t pages from Q and each of the t threads can handle one page.

Even though using multiple threading speeds up the crawler, there is another practical issue that needs to be considered. Crawlers place considerable load on the servers from which it requests the pages. Sending a few hundred requests to a single server at once will cause the servers to slow down considerably and the web site administrators are not happy with this. If too many requests to a server arrive from a single source, then the server may choose to deny the service. Another issue is that that web servers may object to downloading certain pages. To avoid these problems cralwers use *politeness policies.* After sending a batch of requests to a server, the crawler waits for sometime before sending another batch of requests to the same server. Web server administrators maintain a file called called *robots.txt* that explicitly specifies the pages that should be downloaded. For example, `http://www.cs.iastate.edu/robots.txt` page has the following information

```
# Stop good web crawlers (robots) from being mean.
# 2013-Dec-04, MVA.

User-agent: *
Disallow: /cgi-bin/
Disallow: /internal/
Disallow: /outreach/register/

# EOF
```

A polite crawler will first fetch the *robots.txt* page and will not crawl any sites that are *disallowed.* Few other issues that arise in designing a crawler

- Parsing HTML page and extracting links can be non trivial as some pages have malformed links.
- Network interruptions can occur while crawling
- Extracting the text from a web page is challenging as it is not easy to identify the "actual content of a web page", as the actual content may be surrounded by advertisements, etc .

2 Page Rank: Approach 1

Google's Page rank gives a *ranking* to the web pages based on the existing link structure in the web graph. If page p contains a link to page q, then this can be interpreted as "page p thinks that page q has useful information" or "page p is recommending page q". Intuitively, the importance/rank of a page p is determined the following principles.

1. Number of recommendations that a page gets. This is the same as the in degree of the page.
2. Consider the following scenarios: A page q has 100 links and p is one among them. A page q has 5 links and p is one among them. In both scenarios, page q is recommending page p. However, we would like to value the recommendation in the second scenario highly as p is *one among five* whereas in the former scenario page p is *one among 100.* For example, being in page that lists top 5 restaurants is more valuable than being in a page that lists top 100 restaurants. This suggests that the recommendations should be weighted.
3. If a highly ranked page recommends p, then its recommendation should carry more weight than the recommendation from a lowly ranked page.

We will arrive a definition of the page rank based on above guiding principles. Let $G = (V, E)$ be the web graph where $|V| = N$, and M be a matrix representing the graph: $M_{ij} = 1$ if there is a link from j to i; otherwise $M_{ij} = 0$. Let I be a N-dimensional vector whose components are all 1s. Our goal is to define a rank vector $R = \langle r_1, r_2, \cdots r_N \rangle$ where r_i denotes the rank of page i. By considering the first guiding principle, we arrive at the following definition of R

$$R = MI$$

Thus

$$r_i = M_{i1} + M_{i2} + \cdots + M_{in}$$

Thus r_i is simply the number of pages recommending page i.

Now let us consider the second guiding principle. Suppose that page j is recommending page i, thus $M_{ij} = 1$. Let d_j be the total number of recommendations that page j makes. The second principle suggests that the smaller the value of d_j, then the weight of recommendation from page j should be carry more weight. To capture this let us define the matrix $\tilde{M}$ as follows: $\tilde{M}_{ij} = 0$ if there is no link from j to i; otherwise $\tilde{M}_{ij} = 1/d_j$. Here d_j is the out degree of j. By using the first two principles, we arrive at the following definition of R.

$$R = \tilde{M}I$$

Thus

$$r_i = \frac{M_{i1}}{d_1} + \frac{M_{i2}}{d_2} + \cdots + \frac{M_{in}}{d_n} \tag{1}$$

Let us look at Equation 1 closely. Every page is contributing to r_i—the rank of page i. For example page 1 contributes $\frac{M_{i1}}{d_1}$ to the rank of i. Now let us bring in the third principle—contribution from a highly ranked page should carry more weight. Thus if page 1 is highly ranked, then it should contribute "more" to the rank of i. One way to reflect this is to make the contribution of page 1 to r_i to be

$$\frac{M_{i1}}{d_1} \times r_1$$

Thus we arrive at the following

$$r_i = \frac{M_{i1}}{d_1} r_1 + \frac{M_{i2}}{d_2} r_2 + \cdots + \frac{M_{in}}{d_n} r_n \tag{2}$$

Thus we can define the rank vector R as any vector that satisfies the following equation

$$R = \tilde{M} R \tag{3}$$

Recall from Linear Algebra that a vector that satisfies the above equation is known as *eigen vector* corresponding to the *eigen value 1*. The obvious questions are

1. Does R exist?
2. If R exists is it unique?

3 Approach 2: Random Surfer Model

Let us try to arrive at the notion of page rank from a different perspective by considering *random surfer* model. Consider a person who surfs the web as follows: Initially at Step 0, pick a random page and start browsing. Suppose that at step i, the person is at page p, then uniformly at random pick a link from p and go to that page. Imagine that our surfer repeats this process. At any time, What is the probability that (s)he is at a particular page p? For what pages this probability is high? For what pages this probability is low? Intuitively, if many pages point to p, then there is a higher probability of landing at page p. Moreover if many pages on which there is a high probability of landing point to page p, then there is even higher probability that the person lands at page p. Thus we can think of "probability of landing at page p" is the importance of page p, and thus the rank of page p. Let us formalize this. Let P_0 be the initial probability vector, i.e, $P_0 = \langle r_1^0, r_2^0, \cdots r_N^0 \rangle$, where r_i^0 denotes the probability that our surfer is on page i at step 0. Note that this value is $1/N$. Similarly let $P_t = \langle r_1^t, r_2^t, \cdots r_N^t \rangle$ where r_i^t denotes the probability that our surfer is on page i at step t. Note that

$$P_1 = \tilde{M} P_0,$$

$$P_2 = \tilde{M} P_1,$$

$$P_3 = \tilde{M} P_2$$

In general,

$$P_t = \tilde{M} P_{t-1}$$

and

$$P_t = \tilde{M}^{t-1} P_0$$

and so on. Let us look at the sequence $P_0, P_1, \cdots$. Does this converge? I.e., does there exist a t such that P_t is same as P_{t+1}? If so, such a distribution is called *stationary distribution* of the random walk. More formally, P is a stationary distribution if

$$P = \tilde{M} P$$

Apriori, we do not know whether such P exist or not. For a moment suppose that P exists. Then for each page i, then ith component of P is the "eventual" probability that our surfer lands on page i. Thus this is the rank of page i. Compare the above equation with Equation 3. These two are exactly the same! We arrived at the same notion of page rank starting from two different perspectives!

The following is known: If the graph is strongly connected and non-bipartite, then any random walk (starting from any initial distribution) converges to an unique stationary distribution.

However, the web graph is not strongly connected, there are many pages that do not have any outgoing links (known as sinks) and subgraphs from which rest of the graph can not be reached (known a spider traps). Consider a graph with four vertices $V = \{1, 2, 3, 4\}$ and edges $\{\langle 1, 2\rangle, \langle 1, 4\rangle, \langle 1, 3\rangle, \langle 2, 1\rangle, \langle 4, 2, \rangle, \langle 4, 3\rangle\}$. Vertex 3 has no outgoing links, and is a sink. The stationary distribution for this graph is $[0, 0, 0, 0]$. Now consider the same graph with a self loop added at 3, now 3 is a spider trap. The stationary distribution for this graph is $[0, 0, 1, 0]$. Vertex 3 has all the rank, it drains out all ranks from all other vertices. It is easy to create spider traps on web, and existence of spider traps renders the page rank meaningless. We can get around the first the problem by making a simple modifications to to random surfer model: When the surfer arrives at a sink, uniformly a random pick a page (from all possible pages) and go to that page.

Now we have two problems, first is that the graph may not be strongly connected and thus the random walk may not have a stationary distribution, secondly the existence of spider traps. We get around both these problems by modifying the random walk as follows: Recall that at each step of the random walk the surfer uniformly at random picks a link from the current page. Instead the surfer will do the following—Toss a biased coin with probability of head β. If the outcome is tail, then uniformly at random pick a page (from all all possible pages), otherwise uniformly at random pick a link from the current page (if current page does not have any links, then uniformly at random pick a page). Note that this not only avoids spider traps, but also (implicitly) makes the graph strongly connected. It can be shown this random walk converges to an unique stationary distribution.

Recall that the original transition matrix is $\tilde{M}$, and this may have sinks. Let $\tilde{N}$ denote the matrix that is same as $\tilde{M}$ except that replace any column consisting of all zeros (corresponding to sink node), with a column where each entry is $1/N$. Let us define another matrix $\tilde{O}$ as follows: The ijth entry of $\tilde{O}$ is

$$\beta \tilde{N}_{ij} + (1 - \beta)\frac{1}{N}$$

. Now the page rank vector R (which is same as the stationary distribution vector) is the vector satisfying

$$P = \tilde{O} P$$

3.1 Computing Page Rank

An obvious way to compute the page rank vector is to solve the equation $P = \tilde{O}P\rangle$ using Gaussian elimination. However, this is too expensive as Gaussian elimination takes $O(N^3)$ time and is thus impractical on web graphs. Recall the crucial property of the stationary distribution: Starting with any initial distribution and doing a random walk will eventually yield stationary distribution, thus if we consider the following sequence (where U is the uniform distribution)

$$P_0 = U, P_1 = \tilde{O}P_0, P_2 = \tilde{O}P_1, \cdots$$

this sequence will eventually converge to P. Thus suggests the following algorithm: Start with U and iteratively compute P_0, P_0U, P_1U till we find t such that $P_t = P_{t+1}$. Then P_t is the stationary distribution and is the page rank vector. Time taken for this step is $O(tN^2)$, where t is the number of steps needed for the convergence. This algorithm works except for a small caveat: While we know that stationary distribution exists, it only exists in *limit*, and we may not arrive at it after a finite number of steps. Thus in practice, we settle for an approximation.

Given a Matrix M let us define $Norm(M)$ as sum absolute values of all entries of M. We can compute an approximation to the page rank vector by iteratively computing

$$P_0 = U, P_1 = \tilde{O}P_0, P_2 = \tilde{O}P_1, \cdots$$

till we find a t such that $Norm(P_t - P_{t+1}) \leq \epsilon$ for a small ϵ (such as 0.1 or 0.01).

Even though the time taken for this step is $O(tN^2)$ is far less than $O(N^3)$ this is still impractical for large graphs. Now we arrive at a better algorithm. This algorithm simulates the random walk of the surfer. The following algorithm describe one iteration of the algorithm. Suppose P_n is the distribution vector arrived after n steps of the random walk. The algorithm takes P_n as input and outputs P_{n+1}–the distribution vector after $n+1$ steps of the random walk. Given any page p of the graph G, let $P_n(p)$ be the pth component of the vector P_n, i.e, this is the probability that the surfer is on page p after n steps. The algorithm described below simulates one step of the random walk.

```
1. Input $P_n$ and graph $G$.
2. Set $P_{n+1}$ as $[\frac{1-\beta}{N}, \frac{1-\beta}{N}, \cdots, \frac{1-\beta}{N}]$
3. for every page $p$ of the graph $G$ do
    (a) if Number of links in $p$ is not equal to zero then
        • Set $Q$ be the set of all pages that are linked from $p$
        • For every page $q \in Q$ set
              $P_{n+1}(q) = P_{n+1}(q) + \beta\frac{P_n(p)}{|Q|}$
    (b) if Number of links in $p$ is equals zero then
        • For every page $q$ in the graph $G$ set
              $P_{n+1}(q) = P_{n+1}(q) + \beta\frac{P_n(p)}{N}$
```

Let us denote the above algorithm with $\mathcal{A}$. The page rank algorithm described below uses $\mathcal{A}$ as a subroutine. It gets a graph G and approximation parameter ϵ as input.

```
1. Input $G = (V, E)$, $\epsilon > 0$.
2. Set $P_0$ to the uniform probability vector $[\frac{1}{N}, \frac{1}{N}, \cdots, \frac{1}{N}]$.
3. Set $n = 0$ and $P_n$ to $P_0$.
4. Set *converged* to false
5. While (not converged) do
    (a) Compute $P_{n+1}$ by running $\mathcal{A}$ on input $P_n$.
    (b) If $Norm(P_{n+1} - P_n) \leq \epsilon$, then set *converged* to true
    (c) $n = n + 1$.
6. Output $P_n$ as the page rank vector.
```

Finally, to implement the above algorithm we need to set a value of β. The original page rank algorithm takes β as 0.85 and it seems to be a reasonable choice. Observe that the time taken by $\mathcal{A}$ is $O(M + N)$ where M is the number of edges of G. Thus the total time taken by the page rank algorithm is $O(t(M + N))$ where t is the smallest integer for which condition $Norm(P_{t+1} - P_t) \leq \epsilon$ holds. It has been empirically observed that t is pretty small (less than a thousand).