Combinatorial Algorithms for Web Search Engines - Three Success Stories

Monika Henzinger*

Abstract

How much can smart combinatorial algorithms improve web search engines? To address this question we will describe three algorithms that have had a positive impact on web search engines: The PageRank algorithm, algorithms for finding near-duplicate web pages, and algorithms for index server loadbalancing.

1 Introduction.

With an estimated 20% of the work population being online web search has become a huge application, only surpassed by email in the number of users. Being a fairly new research area an obvious question what combinatorial research problems arise in web search engines and how much combinatorial algorithms have already contributed to the success of web search engines. In this paper we will present three "success stories", i.e., three problem areas where combinatorial algorithms have had a positive impact.

The biggest "success story" is certainly the PageRank algorithm [2], which can be viewed as the stationary distribution of a special random walk on the *web graph*. It led to a significant improvement in search quality and gave rise to the creation of the Google search engine. Google currently serves about half of all the web searches. Furthermore, together with the HITS algorithm [12] the PageRank algorithm initiated research in hyperlink analysis on the web, which has become a flourishing area of research. We will briefly describe PageRank in Section 2.

Combinatorial techniques were also successfully applied to the problem of finding near-duplicate web pages. Both the shingling algorithm [4, 5] and a projection-based approach [6] are used by or were created at successful web search engines. We will discuss them and their practical performance in Section 3.

Web search engines build a data structure, called inverted index, to answer user queries. This data structure stores a record for every word in every web page that the web search engine "has indexed", i.e., that it can return as search result. When indexing billions or tens of billions of web pages the inverted index has to be distributed over many machines (called *index servers*) in a way that (1) reduces the number of machines needed, and (2) maximizes the number of user queries that can be answered. The resulting optimization problem and algorithms to solve it are discussed in Section 4.

*Ecole Polytechnique Fédéral de Lausanne (EPFL) & Google

2 Hyperlink Analysis

The first generation web search engines performed quite poorly, especially for broad queries or homepage searches. For a given user query they found hundreds or thousands of documents containing the keywords of the query, but they rarely returned the most relevant documents first. The problem was that they employed techniques that were developed and tested on a very different set of documents, namely on homogeneous, high-quality collections, like newspaper collections. On the web, however, the quality of pages is highly variable and techniques are needed to find the highest-quality pages that contain the query keywords. PageRank [2] is one such technique. It is defined as follows. A *web graph (V,E)* contains one node for every web page and a directed edge from u to v in E iff there is a hyperlink from the page represented by node u to the page represented by node v. Let $n = |V|$ be the total number of web pages, let d be a small constant, like 1/8, let $outdeg(v)$ denote the outdegree of node v and let $PR(u)$ denote the PageRank value of the page represented by node u. Then the PageRank of the page represented by node v is defined recursively as

$$PR(v) = d/n + (1-d) * \sum_{u \text{ with } (u,v) \in E} PR(u)/outdeg(u).$$

Solving this system of linear equations is equivalent to determining the Eigenvector of a suitably chosen matrix. However, in practice, the PageRank values are not computed exactly, but instead the system of linear equation is solved iteratively using roughly a hundred iterations.

The value d/n which is added to PageRank value of every vertex in each iteration is called the *reset value*. One idea for a variant of PageRank is to not give the same reset values to all pages. For example, one can give all pages on a certain topic a high reset value, and set all remaining pages a reset value of 0. This would result in a "topic-flavored" PageRank value. Pushing this idea even further a "personalized PageRank" can

be computed if the reset value of all pages describing best a user's interest are given a positive reset value, and the reset value of all other pages was set to zero. How to achieve this efficiently has been a popular area of research, see e.g. [13]. See the excellent survey by Berkhin [3] for further details, variants, and related work in the area of hyperlink analysis.

3 Finding Near-Duplicate Web Pages

Duplicate and near-duplicate web pages are creating large problems for web search engines: They increase the space needed to store the index, either slow down or increase the cost of serving results, and annoy the users. Thus, algorithms for detecting these pages are needed.

A naive solution is to compare *all* pairs to documents, but this is prohibitively expensive for large datasets. Manber [14] and Heintze [10] were the first to propose algorithms for detecting near-duplicate documents with a reduced number of comparisons. Both algorithms work on sequences of adjacent characters. Brin et al. [1] started to use word sequences to detect copyright violations. Shivakumar and Garcia-Molina [16],[17] continued this work and focused on scaling it up to multi-gigabyte databases [18].

Later Broder et al. [4],[5] and Charikar [6] developed approaches based on solid theoretical foundations. Broder reduced the near-duplicate problem to a set intersection problem. Charikar used random projections to reduce the near-duplicate problem to the problem of determining the overlap in two high-dimensional vectors. Both Broder's and Charikar's algorithms were either developed at or used by popular search engines and are considered the state-of-the-art in finding near-duplicate web pages. We call them *Alg. B* and *Alg. C* and describe them next.

Both algorithms assume that each web page is converted into a sequence of tokens, each token representing a word in the page. For each page they generate a bit string, called *sketch* from the token sequence and use it to determine the near-duplicates for the page.

Let n be the length of the token sequence of a page. For Alg. B every subsequence of k tokens is fingerprinted using 64-bit Rabin fingerprints [15], resulting in a sequence of $n-k+1$ fingerprints, called *shingles*. Let $S(d)$ be the set of shingles of page d. Alg. B makes the assumption that the percentage of unique shingles on which the two pages d and d' agree, i.e. $\frac{|S(d)\cap S(d')|}{|S(d)\cup S(d')|}$, is a good measure for the similarity of d and d'. To approximate this percentage every shingle is fingerprinted with m different fingerprinting functions f_i for $1 \leq i \leq m$ that are the same for all pages. This leads to $n-k+1$ values for each f_i. For each i the smallest of these values is called the *i-th minvalue* and is stored at the page. Thus, Alg. B creates an m-dimensional vector of minvalues. Note that multiple occurrences of the same shingle will have the same effect on the minvalues as a single occurrence. Broder showed that the expected percentage of entries in the minvalues vector that two pages d and d' agree on is equal to the percentage of unique shingles on which d and d' agree. Thus, to estimate the similarity of two pages it suffices to determine the percentage of agreeing entries in the minvalues vectors. To save space and speed up the similarity computation the m-dimensional vector of minvalues is reduced to a m'-dimensional vector of *supershingles* by fingerprinting non-overlapping sequences of minvalues: Let m be divisible by m' and let $l = m/m'$. The concatenation of minvalue $j*l, \ldots, (j+1)*l-1$ for $0 \leq j < m'$ is fingerprinted with yet another fingerprinting function and is called *supershingle*. This creates a supershingle vector. The number of identical entries in the supershingle vectors of two pages is their *B-similarity*. Two pages are *near-duplicates of Alg. B* or *B-similar* iff their B-similarity is at least 2. In order to test this property efficiently *megashingles* are introduced. Each megashingle is the fingerprint of a pair of supershingles. Thus, two pages are B-similar iff they agree in at least one megashingles. To determine all B-similar pairs in a set of pages, for each page all megashingles are created, the megashingles of all pages are sorted, and all pairs of pages with the same megashingle are output. The sketch of a document consists thus of a concatenation of all the megashingles. When applying the algorithm to a large set of web pages usually $m = 84$, $m' = 6$, and $k = 10$ [8].

Alg. C is based on random projections. Its intuition is as follows: Consider a high-dimensional space where every possible token gives rise to a dimension. The *token vector* of a page is a vector in this space where the entry for each token is the frequency of the token in the document. This creates a mapping from pages to points in this space. The probability that the points representing two pages lie on the same side of a random hyperplane through the origin is proportional to the angle between the two points [9], which is equal to the *cosine-similarity* of the two pages. Thus, picking b random hyperplanes and determining the percentage of times where the points lie on different sides gives an unbiased estimator for the cosine similarity.

Alg. C can be implemented as follows. Each token is projected into b-dimensional space by randomly choosing b entries from the range $[-1, 1]$. This creates a *token vector*. Note that the i-th entry in all the token vectors represents the i-th random hyperplane.

The token vector is fixed for the token throughout the algorithm, i.e., it is the same for all pages. For each page a b-dimensional *page vector* is created by adding the token vectors of all the tokens in the token sequence of the page such that the projection of a token that appears j times in the sequence is added j times. The sketch for the page is created by setting every positive entry in the page vector to 1 and every non-positive entry to 0, resulting in a random projection for each page. The sketch has the property that the cosine similarity of two pages is proportional to the number of bits in which the two corresponding projections agree. Thus, the *C-similarity* of two pages is the number of bits their sketches agree on. Two pages are *near-duplicates of Alg. C* or *C-similar* iff the number of agreeing bits in their sketches lies above a fixed threshold t. Note that this is equivalent to saying that the sketches disagree in at most $b - t$ bits. To compute all C-similar pairs efficiently the sketch of every page is split into $b - t + 1$ disjoint pieces of equal length. The piece augmented with its number in the split is called a *subsketch*. Note that if two pages are C-similar, i.e., they disagree in at most $b - t$ bits then they must agree in at least one of the $b - t + 1$ subsketches. Thus, to determine all C-similar pairs the subsketches of all the documents are generated, sorted by subsketch, and the sketch overlap is computed for every pair with the same subsketch. This guarantees that all pairs which agree in at least t are found.

We briefly compare the two algorithms. Both algorithms assign the same sketch to pages with the same token sequence. Alg. C ignores the order of the tokens, i.e., two pages with the same *set* of tokens have the same bit string. Alg. B takes the order into account as the shingles are based on the order of the tokens. Alg. B ignores the frequency of shingles, while Alg. C takes the frequency of tokens into account. For both algorithms there can be false positives (non near-duplicate pairs returned as near-duplicates) as well as false negatives (near-duplicate pairs not returned as near-duplicates.)

A recent evaluation on 1.6B web pages [11] showed that the percentage of correct near-duplicates returned out of all returned near-duplicates, i.e., the *precision*, is about 0.5 for Alg C and 0.38 for Alg. B. To allow for a "fair" comparison both algorithms were given the same amount of space per page, the parameters of Alg. B were chosen as given in the literature (and stated above), and the threshold t in Alg. B was chosen so that both algorithms returned roughly the same number of correct answers, i.e., at the same *recall* level. Specifically, $b = 384$ and $t = 372$.

Both algorithms performed about the same for pairs on the *same* site (low precision) and for pairs on *different* sites (high precision.) However, 92% of the near-duplicate pairs found by Alg. B belonged to the *same* site, but only 74% of Alg. C. Thus, Alg. C found more of the pairs for which precision is high and hence had an overall higher precision.

The comparison study also determined the number of near-duplicates $N(u)$ of each page u. The plot of the distribution of $N(u)$ in log-log scale showed that $N(u)$ follows for both algorithms a power-law distribution with almost the same slope. However, Alg. B has a much "wider spread" around the power law curve than Alg. C. This might be due to the fact that Alg. B computes the sketch of a page based on a random subset of the shingles of the page. Thus, "unlucky" choices of this random subset might lead to a large number of near-duplicates being returned for a page. The sketch computed by Alg. C is based on *all* token occurrences on a page and thus it does not exhibit this problem.

4 Index Server Loadbalancing

Web search engines build a data structure, called inverted index, to answer user queries. This data structure stores a record for every word in every web page that the web search engine "has indexed", i.e., that it can return as search result. When indexing billions or tens of billions of web pages the inverted index has to be distributed over many machines, i.e. *index servers*, in a way that (1) reduces the number of machines needed, and (2) maximizes the number of user queries that can be answered. For serving user requests efficiently it is best to distribute the inverted index by partitioning the set of documents into subsets and by building a complete inverted index for each subset. The latter is called a *subindex*. The challenge is to partition the documents in such a way that (a) each subindex fits on a machine, and (b) the time to serve requests is balanced over all involved machines. Note that each user request has to be sent to every subindex. However, the time to serve a request on a subindex depends on how often the query terms appear in the documents of the subindex. Thus, if, for example, a Chinese query is sent to a subindex that contains only English documents, it will be served very quickly, while it might take a long time on a subindex consisting mostly of Chinese documents. Search engines usually do a pretty good job splitting the documents into subsets such that the ratio of the total time spent serving requests for two different subindices is within a factor of $1 + \alpha$ of each other, where α is less than 1. However, making α arbitrarily close to 0 is hard.

Index servers are usually CPU limited while they have more than enough memory. To improve load-

balancing one can thus place a subindex on multiple machines and then share the requests for the subindex between these machines, sending each request just to one copy of the subindex. Since individual requests take only a short amount of time, usually below a second, this allows for a very fine-grain load-balancing between machines. However, the open questions are which subindices to duplicate, how to assign subindices to index servers, and how to assign request to index servers when there are multiple choices.

This problem can be modeled as follows. We are given m machines $m_1, \ldots, m_m$ and n subindices $f_1, \ldots, f_n$. Each machine m_i can store s_i subindices such that $\sum_i s_i = n + n'$ with integer $n' > 0$. This implies that up to n' subindices can be stored on multiple machines. We are given a sequence of request for subindices. There is a central scheduler which sends requests to machines. A request for subindex f_j must be sent to a machine that stores f_j to be executed on this machine. When machine m_i executes request t and request t was for subindex f_j, then load $l(t, j, i)$ is put on machine m_i. Note that the load depends on the request as well as on the subindex and on the machine. The dependence on t implies that different requests can put different loads on the machines, even though they are accessing the same subindex. The dependence on the machine implies that the machines can have different speeds.

The total *machine load* ML_i on machine m_i is the sum of all the loads put on m_i. The problem is to assign subindices to machines and to design a scheduling algorithm such that the maximum machine load $max_i ML_i$ is minimized. Of course, one can also study optimizing other measures.

Let $FL_j = \sum_j l(t, j)$ be the *index load* of f_j. In the web search engine setting two assumptions can be used:

(1) *Balanced request assumption:* The subindices are usually generated so that for a long enough sequence of requests, the index loads are roughly balanced, i.e., we assume that there is a number L such that $L \leq FL_j \leq (1+\alpha)L$ for all subindices f_j, where α is a small, positive constant. However, neither L nor α are known to the algorithm.

(2) *Small request assumption:* Each individual load $l(t, j, i)$ is tiny in comparison to FL_j. Let $\beta = max_{t,j,i} l(t, j, i)$. We assume that $\beta << FL_j$ for all j.

Assume that for every machine m_i, $s_i \geq (n \text{ div } m)+ 2$. This implies that $n' \geq 2m$. Assume further that the load $l(t, j, i)$ is independent of i, i.e., $l(t, j, i) = l(t, j)$. This implies that all machines have the same speed. We describe next an algorithm that assigns subindices to machines, called the *layout algorithm with imaginary index loads*. The algorithm assumes that every subindex has an *imaginary index load* of 1 and assigns the imaginary index load of every subindex, either whole or in part, to a machine. A subindex is *assigned* to a machine if at least part of its imaginary index load was assigned to the machine. Thus, if the imaginary index load of a subindex is placed on multiple machines, then the subindex is placed on multiple machines. If its imaginary index load was placed completely on one machine, the subindex is placed only on that machine. The algorithm assigns imaginary index loads to machines so that the total imaginary index load placed on the machines are completely balanced, i.e., each machine receives n/m total imaginary index load. When the total imaginary index load placed on a machine is n/m, the machine is called *full.*

In the first step the algorithm places the complete imaginary index load of $(n \text{ div } m)$ arbitrary subindices on each machine. These subindices will only be stored on one machine. If m divides n, all machines are full and the algorithms terminates. Otherwise, no machine is full and the algorithm makes an arbitrary machine the current machine. In the second step the algorithm takes an unassigned subindex and puts as much of its imaginary index load on the current machine as possible, i.e., until the current machine is full or all the imaginary index load of the subindex has been assigned. In the former case an arbitrary non-full machine becomes the current machine, in the later case the second step is repeated if there are still unassigned subindices.

Let ML_* be the maximum machine load achieved for the layout algorithm with imaginary index loads together with a greedy scheduling algorithm and let Opt be the maximum machine load achieved by the optimum algorithm for *any* layout of subindices. Then,

$$ML_* \leq 2(1 + \alpha)Opt + 2\beta$$

[7], i.e., the greedy scheduling algorithm with the layout algorithm with imaginary index loads are within a factor $2(1 + \alpha)$ of optimal.

On real-life search engine data the greedy scheduler with the layout algorithm with imaginary index loads achieved a noticeable improvement over a greedy scheduler that used a layout algorithm that duplicated hand chosen subindices.

5 Conclusions

We presented three successful applications of combinatorial techniques to problems arising in web search engines. Other areas in combinatorial algorithms that are of interest to web search engines are algorithms for processing data streams, lock-free data structures, and external memory algorithms.

References

[1] S. Brin, J. Davis, and H. Garcia-Molina, *Copy Detection mechanisms for digital documents,* Proc. 1995 ACM SIGMOD International Conference on Management of Data, (1995), pp. 398–409.

[2] S. Brin and L. Page, *The anatomy of a large-scale hypertextual Web search engine,* Proc. 7th Int. World Wide Web Conference, 1998, pp. 107–117.

[3] P. Berkhin, *A survey on PageRank computing*, Internet Mathematics, 2(1) (2005), pp. 73–120.

[4] A. Broder. *On the resemblance and containment of documents,* Proc. Compression and Complexity of Sequences '97, 1997.

[5] A. Broder, S. Glassman, M. Manasse, and G. Zweig, *Syntactic clustering of the web,* Proc. 6th International World Wide Web Conference (1997), pp. 393–404.

[6] M. S. Charikar, *Similarity Estimation Techniques from Rounding Algorithms,* Proc. 34th Annual ACM Symposium on Theory of Computing, (2002), pp. 380–388.

[7] P. Duetting, M. Henzinger. Notes.

[8] D. Fetterly, M. Manasse, and M. Najork, *On the Evolution of Clusters of Near-Duplicate Web Pages,* Proc. 1st Latin American Web Congress, 2003, pp. 37–45.

[9] M. X. Goemans and D. P. Williamson. *Improved approximation algorithms for maximum cut and satisfiability problems using semidefinite programming.* JACM 42 (1995), pp. 1115–1145.

[10] N. Heintze, *Scalable Document Fingerprinting,* Proc. 2nd USENIX Workshop on Electronic Commerce, (1996).

[11] M. Henzinger, *Finding Near-Duplicate Web Pages: A Large-Scale Evaluation of Algorithms*, Proc. 29th Annual International Conference on Research and Development in Information Retrieval, 2006, pp. 284–291.

[12] J. Kleinberg, *Authoritative sources in a hyperlinked environment*, JACM, 46 (1999), pp. 604–632.

[13] G. Jeh and J. Widom, *Scaling personalized web search,* Proc. of 12th Int. World Wide Web Conference (2003), pp. 271–279.

[14] U. Manber, *Finding similar files in a large file system.* Proc. of the USENIX Winter 1994 Technical Conference (1994), pp. 1–10.

[15] M. Rabin. *Fingerprinting by random polynomials.* Report TR-15-81, Center for Research in Computing Technology, Harvard University, 1981.

[16] N. Shivakumar and H. Garcia-Molina, *SCAM: a copy detection mechanism for digital documents,* Proc. International Conference on Theory and Practice of Digital Libraries (1995).

[17] N. Shivakumar and H. Garcia-Molina, *Building a scalable and accurate copy detection mechanism,* Proc. ACM Conference on Digital Libraries (1996), pp. 160–168.

[18] N. Shivakumar and H. Garcia-Molina, *Finding near-replicas of documents on the web,* Proc. Workshop on Web Databases (1998), pp. 204–212.