Set Cover at Web Scale

Stergios Stergiou
Yahoo! Labs
Sunnyvale, CA USA
stergios@yahoo-inc.com

Kostas Tsioutsiouliklis
Yahoo! Labs
Sunnyvale, CA USA
kostas@yahoo-inc.com

ABSTRACT

The classic Set Cover problem requires selecting a minimum size subset $\mathcal{A} \subseteq \mathcal{F}$ from a family of finite subsets $\mathcal{F}$ of $\mathcal{U}$ such that the elements covered by $\mathcal{A}$ are the ones covered by $\mathcal{F}$. It naturally occurs in many settings in web search, web mining and web advertising. The greedy algorithm that iteratively selects a set in $\mathcal{F}$ that covers the most uncovered elements, yields an optimum $(1+\ln|\mathcal{U}|)$-approximation but is inherently sequential. In this work we give the first MapReduce Set Cover algorithm that scales to problem sizes of ~ 1 trillion elements and runs in $\log_p \Delta$ iterations for a nearly optimum approximation ratio of $p \ln \Delta$, where Δ is the cardinality of the largest set in $\mathcal{F}$.

A web crawler is a system for bulk downloading of web pages. Given a set of seed URLs, the crawler downloads and extracts the hyperlinks embedded in them and schedules the crawling of the pages addressed by those hyperlinks for a subsequent iteration. While the average page out-degree is ~ 50, the crawled corpus grows at a much smaller rate, implying a significant outlink overlap. Using our MapReduce Set Cover heuristic as a building block, we present the first large-scale seed generation algorithm that scales to ~ 20 billion nodes and discovers new pages at a rate $\sim 4x$ faster than that obtained by prior art heuristics.

1. INTRODUCTION

Set Cover is one of the 21 classic combinatorial problems shown by Karp to be NP-complete in 1972 [1]. Given a collection $\mathcal{F}$ of subsets of a finite set $\mathcal{U}$, the objective is to obtain a minimum subset $\mathcal{A} \subseteq \mathcal{F}$ such that every element in $\mathcal{U}$ belongs to at least one set in $\mathcal{A}$. It is also one of the first problems to be approximated [2] with an approximation ratio of $1 + \ln n$, where $n = |\mathcal{U}|$. This is shown to be optimum as no polynomial-time $(1 - o(1)) \ln n$ approximation algorithm can exist unless NP has slightly superpolynomial time algorithms [3]. Set $k-$Cover is a variation that requires each element in $\mathcal{U}$ to be covered at least k times, while Weighted Set Cover is a variation where each set has an associated weight and the objective is to minimize the sum of the weights of the sets in $\mathcal{A}$.

Johnson's greedy approximation algorithm [2] selects at each step the set that covers the most elements that have not been covered by previous sets and is therefore inherently sequential. In [4] Berger, Rompel and Shor study the problem in a parallel setting and give an NC $O(\log n)$-approximation algorithm. They closely approximate the greedy algorithm by bucketing set cardinalities by factors of $p > 1$ and processing sets within a bucket in parallel. Their techniques lead to an $O(\log^5 M)$-depth, $p \ln n$-approximation randomized algorithm on a PRAM, where $M = \sum_{S \in \mathcal{F}} |S|$. In [5] Blelloch, Peng and Tangwongsan improve upon [4] by obtaining a work-optimum, $O(\log^3 M)$-depth, $p \ln n$ approximation algorithm.

In [6] Chierichetti, Kumar and Tomkins, inspired by the theoretical results of [4], provide the first MapReduce-based algorithm for Max k-Cover, a related problem where the objective is to cover as many elements in $\mathcal{U}$ with at most k sets. Their algorithm closely follows the proof of [4] and requires $O(poly(\epsilon) \log^3 mn)$ MapReduce steps to achieve an $(1 - 1/e - \epsilon)$-approximation randomized algorithm, where $m = |\mathcal{F}|$. Such an algorithm may be an improvement upon the sequential greedy algorithm, but is still not practical for problem instances where $M \sim 1,000,000,000,000$ as in such cases it would require an impractically large ($\sim 100,000$) number of MapReduce steps. In fact their largest dataset has an input size of $M = 72$ million and is comprised of only 14.2 million sets.

In [7] Cormode, Karloff and Wirth present a secondary storage-friendly sequential greedy algorithm for Set Cover. They also adopt the bucketing heuristic of [4] and forgo sampling within each bucket opting instead for a sequential scan of its elements. However during each such scan their algorithm requires moving sets arbitrarily between buckets, which are stored on disk. Such an approach cannot scale to 1 trillion item instances. Furthermore only single-node results are presented, while their largest instance consists of only $M = 5,267,656$ elements.

In this paper we present Set Cover algorithms that scale to problem instances of size $\sim 10^{12}$ and use them to generate seed sets for web crawling, which is one of the important problems addressed by web crawlers.

Web Crawling

A web crawler is a system for batch downloading of web pages. Web crawlers have various applications, the most prominent of which are web search engines, web data min-

KDD'15, August 10-13, 2015, Sydney, NSW, Australia.

DOI: http://dx.doi.org/10.1145/2783258.2783315.

ing and web monitoring. There are currently over 1,500 active crawlers [8]. By some early estimates, crawlers consume up to 40% of the Internet's text web bandwidth [9]. At an abstract level, the web crawling algorithm is straightforward; given a set of *seed URLs* a crawler places them in the *Frontier* structure, from which it iteratively selects URLs, downloads them, extracts the hyperlinks contained in them and adds the new URLs to the Frontier. Web crawling has many engineering challenges, such as scalability, adhering to politeness policies and managing latency, and algorithmic challenges, such as content selection, spam and crawl traps detection, Frontier URLs scheduling, and seed set creation.

Web crawlers are almost as old as the web itself, beginning with Matthew Gray's World Wide Web Wanderer [10] shortly after the launch of NCSA Mosaic, the WWW Worm [11], the RBSE spider [12], WebCrawler [13] and MOMspider [14]. The undocumented crawlers of the first-generation search engines (including Lycos, Infoseek, Excite, AltaVista, and HotBot) followed. The first work that addressed scalability issues was Mike Burner's description of the Internet Archive crawler [15]. Details of Google's first-generation crawler were given by Page and Brin in [16]. Heydon and Najork provided details on the Mercator [17,18], the first documented scalable design which was used in many web research projects [19–23] and adopted by AltaVista in 2001. Another early distributed design was Shkapenyuk and Suel's Polybot web crawler [24]. The IBM WebFountain crawler [25] represented another industrial-strength, fully distributed design. UbiCrawler [26] was the first scalable distributed web crawler that used consistent hashing, allowing for incremental scalability and graceful degradation in the presence of failures. In [27] Lee, Leonard, Wang, and Loguinov shared their experience in designing IRLbot, a single-server web crawler that crawled at a sustained rate of 1,789 pages/sec, downloaded 6.3 billion pages and discovered 41 billion unique nodes in a span of 41 days. Heritrix [28], the crawler currently used by the Internet Archive, and Nutch [29] are two of the most popular open-source crawlers.

The dominant search engines only crawl and index a small fraction of all exposed web pages [30]. The problem of scheduling URLs to be downloaded is therefore crucial. The scheduling policy balances two major goals; (1) coverage, measured either against some prior corpus or via some metric on the collected pages, and (2) freshness, measured via the "staleness" of pages compared to their live versions. Crawling is either a batch or, more commonly, an incremental process that never terminates. Olston and Najork present a thorough survey on scheduling policies in [31].

In [32], Broder, Kumar, Maghoul, Raghavan, Rajagopalan, Stata, Tomkins and Wiener performed an insightful study on the structure of the web. Their findings on web connectivity suggest the existence of five major components; (1) a central strongly connected (SCC) component (28%), (2) a component IN without inlinks from SCC whose outlinks reach SCC (22%), (3) a component OUT without outlinks to SCC that can be reached from SCC (22%), (4) dentril components leaving IN, or entering OUT (22%), and (5) disconnected components (6%.) Their study implies that there exist many node pairs n_i, n_j such that n_j is not reachable from n_i, or only reachable after following hundreds of outlinks. Therefore seed URLs should be selected carefully and multiple seeds may be necessary to ensure good coverage [31]. Ntoulas et al. [33] and Dasgupta at al. [34] studied the creation and retirement of pages and links and found that it is possible to discover 90% of new pages by monitoring links spawned from a small, well-chosen set of old pages, while discovering the remaining 10% requires substantially more effort. Therefore (1) it is possible to deduce a relatively small set of seed URLs that discovers a substantial part of the web, and (2) selecting a random set may leave a significant part of the web undiscovered.

Zheng, Dmitriev and Giles were the first to systematically study the creation of seed lists for web crawling [35]. They proposed a graph-based framework for crawler seed selection, and presented several algorithms within that framework. Evaluation on real web data showed significant improvements over simpler heuristic seed selection approaches. However, many of their algorithms involve calculating the number of nodes reachable within a few hops of a given node and thus, are not scalable. Their experimental results were drawn on a dataset that included 2000 web sites of more than 100 pages each.

MapReduce

In the MapReduce programming paradigm, the basic unit of information is a $(key, value)$ tuple [36, 37]. The input to a MapReduce algorithm is a set of $(key, value)$ tuples while operations on a set of tuples occur in three phases: the map, shuffle and the reduce phase, which we describe hereafter. In the map phase the mapper accepts as input a sequence of tuples and outputs any number of new tuples for each input tuple. Each map operation is stateless which allows for easy parallelization as different inputs for the map can be processed by different computation nodes. During the shuffle phase, all of the values that are associated with an individual key are sent to the same node. The shuffle phase is transparent to the programmer. During the reduce phase the reducer aggregates all of the values associated with a single key k and outputs a multiset of tuples whose key is k. Map and reduce phases are serialized: the reduce phase can only start after all maps have terminated. While each reducer operating on a single key executes sequentially, insofar as the MapReduce paradigm is concerned, reducers operating on different keys can be parallelized. A program in the MapReduce paradigm can comprise many rounds of map/reduce phases executed in a pipelined fashion.

MapReduce is a powerful computational model that has proved successful in enabling large-scale web data mining. Many matrix-based algorithms, such as PageRank [38], have been successfully implemented in the MapReduce model. The authors of [6] suggest three major requirements for efficient MapReduce computations: 1) the number of iterations is at most polylogarithmic in the input size; 2) the output of the map or reduce step should remain linear in the input size. Also, the map and reduce steps should run in time linear in their input sizes; 3) the map/reduce steps should use constant or logarithmic amount of memory.

Our contributions

In this work we develop: (1) an efficient in-memory Set Cover heuristic algorithm that does not require auxiliary data structures of $O(M)$ memory footprint and is guaranteed to complete in $O(m \log m \log_p \Delta + \frac{p}{p-1} M)$ steps while providing a $(p \ln \Delta + 1)$-approximation ratio; (2) a highly scalable MapReduce algorithm for Set Cover that exe-

cutes in $\log_p \Delta$ iterations while providing a $(p \ln \Delta + 1)$-approximation ratio; (3) a LAYERED SET COVER algorithm for generating web crawling seed sets that exploits our MapReduce heuristic and generates seeds that discover new nodes at a significantly faster rate.

Roadmap

The rest of this work is organized as follows: Section 2 provides the necessary definitions. Section 3 presents our main results: an efficient in-memory and a highly scalable MapReduce heuristic algorithm for SET COVER. Section 4 describes our algorithm for generating seed sets for web crawling. Section 5 provides preliminary experimental results. Section 6 concludes this work.

2. PRELIMINARIES

Let $\mathcal{U} = \{1, \ldots, n\}$ be a finite set of n elements and $\mathcal{F}$ a family of subsets of $\mathcal{U}$. Let $m = |\mathcal{F}|$ and $M = \sum_{S \in \mathcal{F}} |S|$. Let $w : \mathcal{F} \to R$ be a weight function on the sets in $\mathcal{F}$.

DEFINITION 2.1. *The k-coverage $cov_k(\mathcal{A})$ of a family of sets $\mathcal{A} \subseteq \mathcal{F}$ is $cov_k(A) = \{x | \sum_{S \in \mathcal{A}} \mathbf{1}_S(x) \geq k\}$.*

where $\mathbf{1}_S(x)$ is the indicator function of set S.

DEFINITION 2.2. *The coverage $cov(\mathcal{A})$ of a family of sets $\mathcal{A} \subseteq \mathcal{F}$ is $cov(\mathcal{A}) = cov_1(\mathcal{A}) = \cup_{S \in \mathcal{A}} S$.*

DEFINITION 2.3 (MIN SET COVER). *$\mathcal{A} \subseteq \mathcal{F}$ is a minimum set cover if $cov(\mathcal{A}) = cov(\mathcal{F})$ and*

$$OPT = |\mathcal{A}| = \min_{\mathcal{S} \subseteq \mathcal{F}, cov(\mathcal{S}) = cov(\mathcal{F})} |\mathcal{S}|.$$

DEFINITION 2.4 (MAX-k-COVER). *$\mathcal{A} \subseteq \mathcal{F}$ is a max-k-cover if $|\mathcal{A}| \leq k$ and $|cov(\mathcal{A})| = \max_{\mathcal{S} \subseteq \mathcal{F}, |\mathcal{S}| \leq k} |cov(\mathcal{S})|$.*

DEFINITION 2.5 (MIN SET k-COVER). *$\mathcal{A} \subseteq \mathcal{F}$ is a minimum k-set cover if $cov_k(\mathcal{A}) = cov_k(\mathcal{F})$ and*

$$|\mathcal{A}| = \min_{\mathcal{S} \subseteq \mathcal{F}, cov_k(\mathcal{S}) = cov_k(\mathcal{F})} |\mathcal{S}|.$$

DEFINITION 2.6 (WEIGHTED SET COVER). *$\mathcal{A} \subseteq \mathcal{F}$ is a weighted set cover if $cov(\mathcal{A}) = cov(\mathcal{F})$ and*

$$\sum_{S \in \mathcal{A}} w(S) = \min_{\mathcal{S} \subseteq \mathcal{F}, cov(\mathcal{S}) = cov(\mathcal{F})} \sum_{S \in \mathcal{S}} w(S).$$

3. APPROXIMATE SET COVER

We begin by describing the classic GREEDY set cover heuristic. We identify performance bottlenecks as well as scalability issues. We then proceed to describe FGREEDY, an efficient in-memory heuristic that guarantees the same approximation as GREEDY. We further refine FGREEDY to RFGREEDY by limiting the amount of access allowed to $\mathcal{F}$ and analyze the impact on the approximation guarantee. Equipped with these results, we present PGREEDY, a MapReduce SET COVER approximation algorithm designed to scale to 10^{12} input size and SPGREEDY, a simplification over PGREEDY that maintains a similar approximation guarantee.

Algorithm 1 GREEDY SET COVER approximation

Require: a family $\mathcal{F}$ of subsets of $\mathcal{U} = \{1, \ldots, n\}$
1: $\mathcal{A} = \emptyset$
2: $C = \cup_{S \in \mathcal{F}} S$
3: **while** $C \neq \emptyset$ **do**
4: $\quad S = \arg\max_{S \in \mathcal{F}} |S \cap C|$
5: $\quad \mathcal{A} = \mathcal{A} \cup \{S\}$
6: $\quad C = C \backslash S$
7: **return** $\mathcal{A}$

3.1 GREEDY

Johnson's classic GREEDY approximation algorithm is shown in Algorithm 1. At each iteration, the algorithm selects the set that has the largest overlap with the set of currently uncovered elements. After each selection S, all elements that have been covered by S are removed from the remaining sets. This operation is typically performed by maintaining an inverted index, mapping each element to the sets it is contained into. This auxiliary structure is as large as the problem input itself. Upon selecting a new set S on line 4 in Algorithm 1, the new ordering of the sets needs to be computed. This is an expensive operation as a particular element may be contained in a large number of sets. Also, updating the position of a particular set within the ordering maybe not be useful as this set may never be selected by the algorithm. Moreover, the ordering itself requires additional space, albeit $O(m)$ instead of $O(M)$.

3.2 FGREEDY

In this section we present an in-memory heuristic that achieves the approximation ratio of GREEDY but does not require constructing an inverted index. Even though it requires the construction of a heap of m elements, the space required for this is as much as the space required for maintaining an ordering on the sets in Algorithm 1. The algorithm first computes the cardinalities of all the sets in the input and builds a max-heap of pointers to the sets, ordered by their cardinalities. The algorithm also maintains a set of uncovered elements, initialized to $\mathcal{U}$, in the form of a bit-vector. At each iteration, the set on the top of the heap is examined. The current number of uncovered elements within it, and subsequently, its position in the heap, is updated. If it remains the top element in the heap after updating, then it is selected as part of the solution. In Lemma 3.1 we show that FGREEDY indeed achieves the same approximation ratio as GREEDY.

LEMMA 3.1. FGREEDY *yields a $\ln \Delta + 1$ approximation*

PROOF. We will show that FGREEDY makes the same set choices as GREEDY. A set T is selected as part of the solution only on line 9 in algorithm 2. At that point, the number of new elements contributed by T with respect to the current coverage C is accurate, since a *decrease_key* operation has been applied on line 6 that has repositioned T in the heap based on its new key $|T \cap C|$. It is also the maximum cardinality set in the heap H. Any prior set selection would have reduced the coverage of remaining sets in H, therefore T is indeed the set which maximizes $T \cap C$. Let us now assume that T should be placed in $\mathcal{A}$ but the algorithm does not select it. In this case, the heap-down operation on line 6 has pushed the set down from the root of H. However, as T is indeed part of the GREEDY solution, all sets T' that

Algorithm 2 fGreedy Set Cover approximation

Require: a family $\mathcal{F}$ of subsets S_j of $\mathcal{U} = \{1, \ldots, n\}$
1: $\mathcal{A} = \emptyset$
2: $C = \cup_{S \in \mathcal{F}} S$
3: H = max-heap on the sets S_j ordered by $|S_j|$
4: **while** $C \neq \emptyset$ **do**
5: T = find_max(H)
6: decrease_key(H, T, $|T \cap C|$)
7: **if** $T ==$ find_max(H) **then**
8: delete_max(H)
9: $\mathcal{A} = \mathcal{A} \cup \{T\}$
10: $C = C \backslash T$
11: **else**
12: $T = T \cap C$
13: **return** $\mathcal{A}$

precede it in H will have $|T' \cap C| < |T \cap C|$, and therefore will be pushed lower than T in subsequent iterations of the algorithm without being selected. □

3.3 rfGreedy

The fGreedy heuristic does not require the construction of an inverted index thus almost halving the memory requirements. It also avoids updating the relative ordering of all the sets at each iteration, opting for correcting it only when necessary. Its performance is dependent on whether a set is examined multiple times during the execution of the algorithm on average. In this section we present rfGreedy, a refinement on fGreedy that bounds the number of times each specific set can be discovered during the execution of the algorithm, at the cost of selecting a slightly less optimal set at each iteration.

Given an approximation factor q, the algorithm selects the top element T in the heap to be placed in the result, even if it would no longer maintain its position at the top of the heap, so long as the uncovered elements $T \cap C$ are at least a fraction $1/q$ of the uncovered elements at the time T was last examined. In Lemma 3.2, we show that rfGreedy parses the input at most $\frac{2q-1}{q-1}$ times. In Lemma 3.3, we show that it is an $O(m \log m \log \Delta + \frac{q}{q-1}M)$-time algorithm. Finally, in Lemma 3.4 we show that rfGreedy is a $(q \ln \Delta + 1)$-approximation algorithm.

Lemma 3.2. rfGreedy *accesses the input M at most $\frac{2q-1}{q-1}$ times*

Proof. Constructing the heap on line 3 in algorithm 3 requires computing the initial cardinalities of all the sets in $\mathcal{F}$, contributing M to the total access of the problem input. Let us now upper bound the number of accesses of each individual set $S \in \mathcal{F}$. S can only be accessed when it is at the top of H. At this point, either S is selected to be placed in $\mathcal{A}$ or its cardinality has been reduced to $1/q$ of the one computed the last time it was placed or moved in the heap. Therefore, a total of $\sum_{j \geq 0} |T|/q^j < \frac{q}{q-1}|T|$ elements of T will be accessed. Aggregating over all sets, the total element accesses is upper bounded by $\frac{q}{q-1}M$. □

Lemma 3.3. rfGreedy *completes in $O(m \log m \log_q \Delta + \frac{q}{q-1}M)$ steps*

Proof. Each set will be examined in the heap at most $\log_q \Delta$ times while each heap-down operation costs $O(\log m)$.

Algorithm 3 rfGreedy Set Cover approximation

Require: a family $\mathcal{F}$ of subsets S_j of $\mathcal{U} = \{1, \ldots, n\}$
Require: $q > 1$
1: $\mathcal{A} = \emptyset$
2: $C = \cup_{S \in \mathcal{F}} S$
3: H = max-heap on the sets S_j ordered by $|S_j|$
4: **while** $C \neq \emptyset$ **do**
5: T = find_max(H)
6: $s = |T|$
7: $s' = |T \cap C|$
8: **if** $qs' \geq s$ **then**
9: delete_max(H)
10: $\mathcal{A} = \mathcal{A} \cup \{T\}$
11: $C = C \backslash T$
12: **else**
13: decrease_key(H, T, s')
14: $T = T \cap C$
15: **return** $\mathcal{A}$

By Lemma 3.2, a total of $O(\frac{q}{q-1}M)$ probes to the input will be performed. □

Lemma 3.4. rfGreedy *is a $(q \ln \Delta + 1)$-approximation algorithm*

Proof. Let us consider the sequence of sets

$$C, C_1, C_2, \ldots, C_{TOT-1}, \emptyset$$

of elements remaining to be covered, obtained after each selection of rfGreedy on line 10. After the j-th selection, there remain $|C_j|$ elements to be covered.

The optimum solution contains OPT sets. Therefore before the j-th selection, there exists a set that contains at least $|C_j|/OPT$ elements. Since Greedy selects the largest set, its selection will contain at least that many elements.

Furthermore, rfGreedy selects a set that contains at least $1/q$ of the elements of the set that would have been selected by Greedy. Therefore, rfGreedy selects a set that covers at least $\frac{|C_j|}{qOPT}$ elements.

We will first compute the number of selections t that rfGreedy makes until at most OPT elements remain to be covered. It holds:

$$\begin{aligned} n(1 - \frac{1}{qOPT})^t &\geq OPT \\ n(1 - \frac{1}{qOPT})^{\frac{qOPTt}{qOPT}} &\geq OPT \\ n(1/e)^{\frac{t}{qOPT}} &\geq OPT \end{aligned}$$

The inequality holds because $f(x) = (1 - 1/x)^x$ is a monotonically increasing function whose limit is $1/e$. Then:

$$\begin{aligned} e^{\frac{t}{qOPT}} &\leq \frac{n}{OPT} \\ t &\leq qOPT \ln \frac{n}{OPT} \\ t &\leq OPTq \ln \Delta \end{aligned}$$

This holds because the optimum solution needs to contain at least n/Δ sets for it to be a cover of $\mathcal{U}$.

Once the number of remaining elements to be covered is reduced to OPT, each subsequent selection will cover at

Algorithm 4 pGreedy Set Cover approximation

Require: a family $\mathcal{F}$ of subsets S_j of $\mathcal{U} = \{1, \ldots, n\}$
Require: a sequence of partition points $p_1 < \ldots < p_k$ for $[1, \Delta]$
Require: $q > 1$
1: $\mathcal{A} = \emptyset$
2: $C = \cup_{S \in \mathcal{F}} S$
3: **for** $L = k..1$ **do**
4: $\quad \mathcal{S}_L = \{S \in \mathcal{F} \mid |S \cap C| \geq p_L\}$
5: $\quad H =$ max-heap on the sets $S \in \mathcal{S}_\mathcal{L}$ ordered by $|S \cap C|$
6: $\quad$ **while** $C \neq \emptyset$ and $\mathcal{S}_L \neq \emptyset$ **do**
7: $\quad\quad T =$ find_max(H)
8: $\quad\quad s = |T|$
9: $\quad\quad s' = |T \cap C|$
10: $\quad\quad$ **if** $qs' \geq s$ **then**
11: $\quad\quad\quad$ delete_max(H)
12: $\quad\quad\quad \mathcal{S}_L = \mathcal{S}_L \backslash \{T\}$
13: $\quad\quad\quad \mathcal{A} = \mathcal{A} \cup \{T\}$
14: $\quad\quad\quad C = C \backslash T$
15: $\quad\quad$ **else if** $s' \geq p_L$ **then**
16: $\quad\quad\quad$ decrease_key(H, T, s')
17: $\quad\quad\quad T = T \cap C$
18: $\quad\quad$ **else**
19: $\quad\quad\quad$ delete_max(H)
20: $\quad\quad\quad \mathcal{S}_L = \mathcal{S}_L \backslash \{T\}$
21: **return** $\mathcal{A}$

least one additional element. Therefore $TOT \leq t + OPT$, which implies that:

$$TOT \leq OPT(q \ln \Delta + 1)$$

This completes the proof. □

3.4 pGreedy

While rfGreedy provides an efficient in-memory heuristic, the focus of this work is set cover instances that are many times larger than what can fit in a single-server memory. rfGreedy provides an essential building block for our very large scale Set Cover algorithm. While it minimizes the memory requirements for auxiliary data structures, it still expects the input to be present in memory. In this section we present pGreedy, a MapReduce algorithm that uses rfGreedy as its basic building block. We draw our inspiration from [4, 6, 7], in particular adopting a variation of the out-degree bucketing approach in order to split the input into chunks that can be handled by rfGreedy. The algorithm is presented in Algorithm 4. We split the input according to a sequence of k points $p_1 < \ldots < p_k$ that partition $[1, \Delta]$ into ranges $[p_1, p_2), \ldots, [p_{k-1}, p_k), [p_k, \Delta]$. The input sets whose current out-degree is within a particular range are processed by rfGreedy. Ranges are processed sequentially from $[p_k, \Delta]$ to $[p_1, p_2)$. We opted in favor of a more general bucketing schema than $p_j = p^{j-1}$ as found in [4,6,7], as this allows us to fine-tune the overall execution time of the algorithm by selecting appropriate buckets that balance the execution times of each individual node. We also note that our algorithm selects sets differently from [6] and [7] in a non-trivial way. Specifically, it is possible for a set to be selected (on line 13 in Algorithm 4) at iteration $L = j$, even if the number of new elements it covers is smaller than the lower bound p_j for the particular bucket. Thus, the algorithm tends to treat each set within the bucket similarly, while in previous approaches sets whose cardinalities were closer to the lower bound of their respective buckets would tend to be placed more easily at a lower bucket. Using rfGreedy as a building block also allows us to separate the bucketing from the approximation guarantee. In our algorithm, bucketing is present such that it splits the input into chunks that can fit in memory (space trade-off), while the approximation ratio of the overall algorithm is still dictated by rfGreedy (time trade-off.) In Lemma 3.5 we show that pGreedy maintains the same approximation ratio as rfGreedy.

Algorithm 5 spGreedy Set Cover approximation

Require: a family $\mathcal{F}$ of subsets S_j of $\mathcal{U} = \{1, \ldots, n\}$
Require: a sequence of partition points $p_1 < \ldots < p_k$ for $[1, \Delta]$
1: $\mathcal{A} = \emptyset$
2: $C = \cup_{S \in \mathcal{F}} S$
3: **for** $L = k..1$ **do**
4: $\quad \mathcal{S}_L = \{S \in \mathcal{F} \mid |S \cap C| \geq p_L\}$
5: $\quad$ **while** $C \neq \emptyset$ and $\mathcal{S}_L \neq \emptyset$ **do**
6: $\quad\quad$ pick a set $T \in \mathcal{S}_L$
7: $\quad\quad \mathcal{S}_L = \mathcal{S}_L \backslash \{T\}$
8: $\quad\quad$ **if** $|T \cap C| \geq p_L$ **then**
9: $\quad\quad\quad \mathcal{A} = \mathcal{A} \cup \{T\}$
10: $\quad\quad\quad C = C \backslash T$
11: **return** $\mathcal{A}$

Lemma 3.5. pGreedy *is a* $(q \ln \Delta + 1)$*-approximation algorithm when* $q \leq \min_j \frac{p_j}{p_{j-1}}$.

Proof. The proof follows that of lemma 3.4. We note that since the heap H allows the examination of the sets in $\mathcal{S}_L$ in non-increasing order, the arguments of lemma 3.4 still hold even though H does not contain all sets of $\mathcal{F}$. □

3.5 spGreedy

In this section we present spGreedy, a further simplification of our MapReduce algorithm. We observe that, if $q \geq \max_j \frac{p_j}{p_{j-1}}$ then an approximation ratio of $\max_j \frac{p_j}{p_{j-1}}$ can be achieved without the overhead of a max-heap. This is because any set selected during the processing of range $[p_{j-1}, j_j)$ cannot be more than p_j/p_{j-1} smaller than the optimum set that could have been selected at that point. This algorithm is more similar to the algorithms in [4,6] but differs from the one in [7]. In the latter, sets are placed in buckets that are stored on disk. When a set is rejected (as in our case happens on line 8 in Algorithm 5,) it is moved to a lower bucket. In our approach, we simply ignore the set altogether, as it will be placed at the appropriate partition of $[1, \Delta]$ during the next MapReduce cycle (specifically, during the next iteration of L.) In Lemma 3.6, we bound the approximation ratio of spGreedy based on its partition points. If the partition points are selected as powers of a factor p, Theorem 3.7 formally states the algorithm's approximation ratio and MapReduce iterations.

Lemma 3.6. spGreedy *is a* $(p \ln \Delta + 1)$*-approximation algorithm where* $p = \max_j \frac{p_j}{p_{j-1}}$.

Proof. The proof follows that of lemma 3.4. We note that although there is no heap to maintain an order on the

Algorithm 6 LAYERED SET COVER

Require: a family $\mathcal{F}$ of subsets of $\mathcal{U} = \{1, \ldots, n\}$
Require: a subset $\mathcal{L}_0 \subseteq \mathcal{F}$
Require: a depth D
1: $\mathcal{A} = \emptyset$
2: $\mathcal{C} = \mathcal{F}$
3: compute families $\mathcal{L}_1, .., \mathcal{L}_D$ by Breadth-First traversal
4: **for** $I = D..1$ **do**
5: $SC_I = \mathcal{L}_I \cap \mathcal{C}$
6: **for** $J = (I-1)..0$ **do**
7: compute SET COVER SC_J of SC_{J+1}
8: $\mathcal{A} = \mathcal{A} \cup SC_0$
9: remove all covered sets from $\mathcal{C}$
10: **return** $\mathcal{A}$

sets, if a set S is selected on line 6 in algorithm 5 at iteration $L = j$, then there does not exist a set S' for which $|S'| > |S|p_{j+1}/p_j$ as it would have been selected at a previous iteration. □

THEOREM 3.7. *Let $p_j = p^{j-1}$. Then* spGREEDY *is a $(p \ln \Delta + 1)$-approximation algorithm that completes in $\log_p \Delta$ MapReduce iterations.*

3.6 MapReduce Realization

We now describe implementation details for algorithms pGREEDY and spGREEDY, which use the Hadoop Pipes API and are implemented in C++11. Set elements are represented as integers from a contiguous range $[1..m]$. A common building block that is utilized by both the mappers and the reducers is the representation of a cover. Cover C is represented by a bit-vector which is stored as a binary file BV in HDFS. The value of the i-th bit in BV denotes the presence of element i in C.

Each iteration of the algorithms corresponds to a different MapReduce job. At the beginning of each iteration, BV is copied in parallel onto the local storage of all map nodes. During the map phase, BV is read-only memory-mapped. The map phase is a filter on the sets, allowing only those sets to pass that contain a specific number of undiscovered elements.

During the reduce phase, BV is copied in memory and is updated as per the algorithms' specifications. The reducer emits a sequence of set IDs and updates the memory copy of BV to reflect the updates on C. Upon termination, BV is stored in HDFS, ready to be redistributed to all the mappers in the next iteration. BV is comprised of m bits, and therefore is comparatively small. For instance, a 10^{10} sets input only requires 2.25GB of space for BV.

3.7 Set Multicover Variation

A natural variation of SET COVER requires each element in $\mathcal{U}$ to be covered by more than a specific number $k > 1$ of sets. We observe that our results extend to this variation. However, in order to support the multicover semantics, each bit in BV needs to be replaced with a modulo-k counter which requires $\lceil \log(k+1) \rceil$ bits. For small values of k this may still yield a practical solution.

3.8 Weighted Set Cover Variation

Our results can be extended to the WEIGHTED SET COVER variation (with a caveat: all $\ln \Delta$ terms would be relaxed to $\ln m$.) This requires the additional transfer of the weight of each set during the map phases. As a minor optimization, we note that if a total ordering on the weights can be imposed, then an alternative to transferring extra information would be to encode the set's weight via its position in BV.

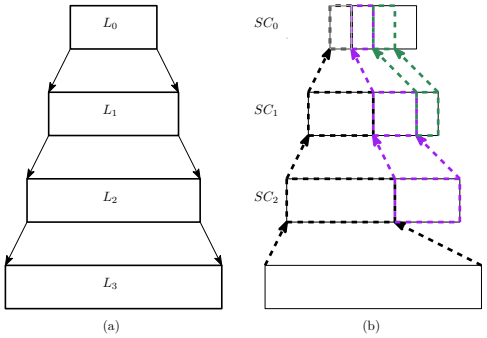

Figure 1: Pictorial description of the Layered Set Cover algorithm. (a) during the first phase, a set of "good" nodes is obtained via simple filter-based heuristics as L_0. Subsequent layers contain the nodes obtained via a Breadth-First traversal; (b) during the second phase, the resulting seed set is incrementally constructed via 3 sub-phases, denoted with black, purple and green color respectively.

4. LAYERED SET COVERING

In this section we describe how our SET COVER heuristics can be used to calculate web crawling seed sets. As the authors of [35] note, a natural formulation for the problem of obtaining seed sets from a web corpus is MAX k-COVER, where each set input S_j corresponds to a node n_j in the graph, and contains as elements all nodes that can be reached from n_j by a D-level deep breadth-first traversal. This approach however, is applicable to very small datasets and is impossible to implement at scale, as the input quickly blows up to an unmanageable size where each set includes many millions of elements.

Instead, we propose an alternative formulation based on SET COVER. Given a corpus IN, a subset L_0 of IN based on some quality criteria and a depth D, we perform a D-level deep Breadth-First traversal, thus obtaining node sets $L_1, L_2, .., L_D$. In the context of set covering, a node n_j in L_i corresponds to a set that includes all nodes that are pointed to by the outlinks of n_j. We then obtain a set cover SC_{D-1} of L_D as a subset of L_{D-1}, followed by a set cover SC_{D-2} of SC_{D-1} as a subset of L_{D-2}. Eventually we obtain SC_0 which is part of the final seed set. If a D-depth Breadth-First traversal were to be executed from SC_0, at least all nodes in L_D would be discovered. We then remove all covered nodes and repeat the algorithm. Since all nodes at level D have been covered, we only traverse the graph until depth $D-1$. We continue in this fashion until all nodes in $L_1, .., L_D$ have been covered. The LAYERED SET COVER algorithm is depicted in Figure 1 and formally presented in Algorithm 6.

5. EXPERIMENTAL RESULTS

We explore the performance of our MapReduce SET COVER heuristic as well as our LAYERED SET COVER heuristic. We

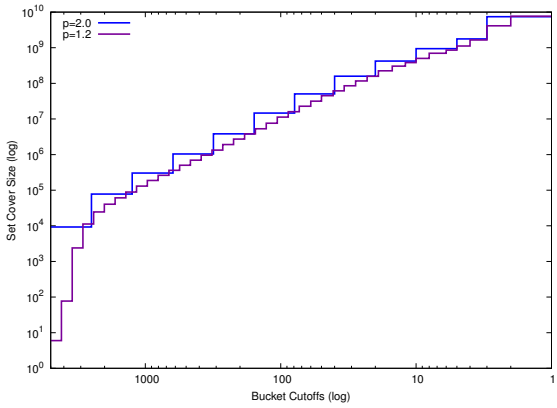

Figure 2: Performance of spGreedy as a function of approximation factor p. The resulting cover sizes for $p = 1.2$ and $p = 2$ are 7,655,425,042 and 7,421,503,788 respectively.

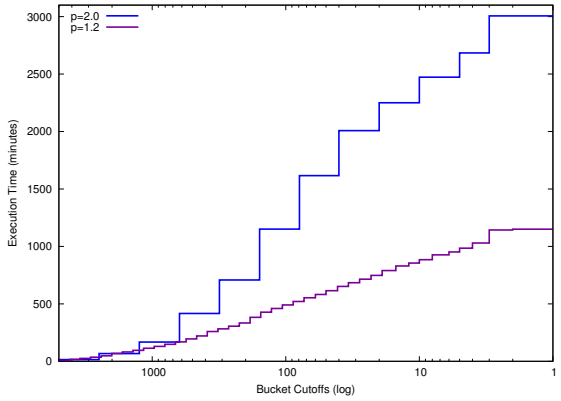

Figure 3: Relative execution times of spGreedy for $p = 1.2$ and $p = 2$. The algorithm completes all iterations in 1150 and 3006 minutes respectively.

draw our results on a 1,500-node cluster whose nodes comprise 2x Intel E5-2620 processors and 64GB of memory (see Table 1.) Our dataset is a large subset of the crawled web. It includes 20.6 Billion pages and approximately 918 Billion outlinks. The average out-degree is 44.63 while the maximum out-degree is limited to 5,000. The disk footprint of the graph is 12TB, stored in 1,000 part files on HDFS (see Table 2.)

Subsequently, we examine the sensitivity of the algorithm on the approximation parameter p by obtaining covers for the whole graph for $p = 1.2$ and $p = 2$. We show the runtime performance of the algorithm for the same parameters.

Following that, we describe the first experiment that exploits our Set Cover heuristic. We show that it is possible to start from an extremely small subset of the graph (11M pages) and discover the majority of the graph within two BFS hops.

We then examine the performance of a seed list generated by our Layered Set Cover heuristic, by comparing it against a high-quality, user-generated seed list. We show that the Set Cover-based seed list yields 3.84 times more pages, 4 hops away from the seed list.

Finally, we report that our Layered Set Cover-based seed list resulted in a significant lift of the median of a PageRank-based metric by $2.32x$ over all pages discovered.

5.1 Cover Size Relative to p

In Figure 2 we present the cardinality of the resulting covers of the spGreedy algorithm as a function of the approximation factor p, for $p = 1.2$ and $p = 2$. For a value of p that tends to 1, the algorithm will perform exactly as Greedy, thereby allowing us to draw comparisons with the classic in-memory heuristic even though it is not possible to execute it for inputs of this size. We observe that the resulting cover sizes are almost identical (they are within 2% of each other.) This result shows that the algorithm is stable for different small values of p.

Table 1: Cluster Specifications

Property	Value
Nodes	1500
CPUs per Node	2x Intel(R) Xeon(R) CPU E5-2620
Memory per Node	64GB

5.2 Execution Times Relative to p

In Figure 3 we present the execution times of all the individual iterations of spGreedy for $p = 1.2$ and $p = 2$. We observe that the performance of the algorithm is significantly better for $p = 1.2$, a fact that may seem counter-intuitive at first, as the approximation guarantee is tighter in this case. However this difference is easily explained: As p tends to 1, the aggregate output of the mappers is smaller and thus, the single reducer is presented with a smaller amount of information to process. Moreover, the more tuples the reducer is called to process, the higher the chances that subsequent tuples will be rejected as they fail to pass the bucket membership test. From an engineering perspective, the optimum scenario occurs when the time spent to complete all mappers is equal to the time spend on the single reducer, on average across all iterations. This equality is well approximated when $p = 1.2$. We note that with our heuristic, it is indeed possible to cover a significant subset of the web graph within only 19 hours. Moreover, our algorithm scales essentially linearly with the input graph size and can be applied to larger graphs as well.

5.3 Layered Set Cover

We ran Algorithm 6 on the input graph for $D = 4$. A total of 10 executions of spGreedy were performed. We used the seed set result $\mathcal{A}$ as a starting point for a depth-4 Breadth-First traversal. Similarly, we performed a depth-4 traversal starting from a manually-curated, high-quality seed set of similar size to $\mathcal{A}$. We present the results in Figure 4. We observe a significant increase of the frontier size as

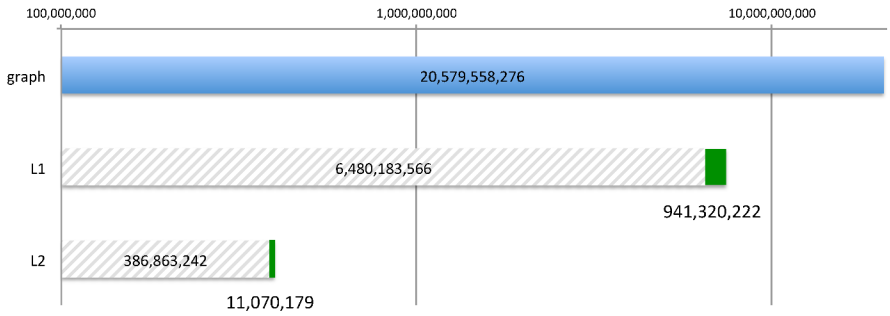

Figure 5: 2-Level Set Cover. Sizes are presented in log-scale. The web graph comprises 20.6 billion pages. L1 is its cover, and contains 941M pages that discover at least 3 unique pages, and 6.5B low-yield pages that only cover at most 2 uncovered pages. L2 is the Set Cover of the 941M-page subset of L1. It contains 11M pages that discover at least 3 unique pages, and 387M low-yield pages that only cover at most 2 uncovered pages.

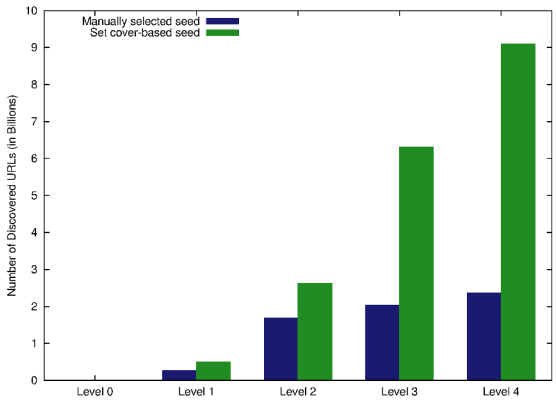

Figure 4: Relative Coverage Increase of seeds generated by the Layered Set Cover algorithm.

the traversal evolves. In fact, after 4 hops, seed set $\mathcal{A}$ has discovered $3.84x$ more pages compared to the user-selected seed set.

In practice, this increase can be exploited in two major ways: First, it allows for crawling at the same target speed with reduced resources. More importantly, it can be used to increase freshness of a corpus, as a fixed-size corpus can be discovered, and thus re-traversed, more frequently.

Table 2: Web Graph Properties

Property	Value
URLs	20,579,558,276
Aggregate Outlinks	918,377,573,523
Maximum Out-degree	5,000 (capped)
Average Out-degree	44.63
Graph Storage Footprint	12TB
Part Files	1,000

5.4 2-Level Set Cover of the Web Graph

We study the set cover of the whole graph and observe that it contains 941M high-yield pages that discover at least 3 uncovered pages, and 6.5B low-yield pages that only cover at most 2 uncovered pages. We then use the 941M high-yield subset and execute spGreedy on it to obtain its cover. This cover in turn contains 11M high-yield pages and 387M low-yield pages.

In essence, we have thus discovered a very small subset of pages that can extract the majority of the web graph after crawling only 2-hops out.

6. CONCLUSION

We developed efficient MapReduce approximation algorithms for Set Cover and showed that they scale to input sizes of approximately 1 trillion elements. We designed an algorithm for creating seed sets for web crawling that utilizes our MapReduce Set Cover heuristic. Seed sets generated by our algorithm were shown to discover new nodes at a rate $\sim 4x$ faster. This marks the first time that Set Cover algorithms have been executed at such large scale.

Acknowledgements

We would like to thank the Yahoo Search Team and in particular, Yoram Arnon and Daeho Baek for their insightful comments and support of this work.

7. REFERENCES

[1] R. M. Karp, "Reducibility Among Combinatorial Problems," *Complexity of Computer Computations*, pp. 85 – 103, 1972.

[2] D. S. Johnson, "Approximation algorithms for combinatorial problems," *Journal of Computer and System Sciences*, vol. 9, no. 3, pp. 256 – 278, 1974.

[3] U. Feige, "A Threshold of $\ln n$ for Approximating Set Cover," *J. ACM*, vol. 45, no. 4, pp. 634–652, 1998.

[4] B. Berger, J. Rompel, and P. W. Shor, "Efficient NC algorithms for set cover with applications to learning and geometry," *Journal of Computer and System Sciences*, vol. 49, no. 3, pp. 454 – 477, 1994. 30th IEEE Conference on Foundations of Computer Science.

[5] G. E. Blelloch, R. Peng, and K. Tangwongsan, "Linear-work Greedy Parallel Approximate Set Cover and Variants," in *Proceedings of the Twenty-third Annual ACM Symposium on Parallelism in Algorithms and Architectures*, pp. 23–32, ACM, 2011.

[6] F. Chierichetti, R. Kumar, and A. Tomkins, "Max-Cover in Map-Reduce," in *Proceedings of the 19th International Conference on World Wide Web*, pp. 231–240, ACM, 2010.

[7] G. Cormode, H. Karloff, and A. Wirth, "Set Cover Algorithms for Very Large Datasets," in *Proceedings of the 19th ACM International Conference on Information and Knowledge Management*, pp. 479–488, ACM, 2010.

[8] "List of spiders and crawlers."

[9] S. Bal and R. Nath, "Filtering the Web Pages that are not Modified at Remote Site Without Downloading using Mobile Crawlers," *Information Technology Journal*, vol. 9, no. 2, pp. 376–380, 2010.

[10] M. Gray, "Internet growth and statistics: Credits and background," 1993.

[11] O. A. McBryan, "Genvl and wwww: Tools for taming the web," in *Proceedings of the first international World Wide Web conference*, vol. 341, 1994.

[12] D. Eichmann, "The rbse spider-balancing effective search against web load," in *Proc. 1st WWW Conf*, 1994.

[13] B. Pinkerton, "Finding what people want: Experiences with the webcrawler," in *Proceedings of the Second International World Wide Web Conference*, vol. 94, pp. 17–20, 1994.

[14] R. T. Fielding, "Maintaining distributed hypertext infostructures: Welcome to MOMspider's web," *Computer Networks and ISDN Systems*, vol. 27, no. 2, pp. 193–204, 1994.

[15] M. Burner, "Crawling towards eternity: Building an archive of the World Wide Web," *Web Techniques Mag.*, vol. 2, no. 5, 1997.

[16] S. Brin and L. Page, "The anatomy of a large-scale hypertextual web search engine," *Computer networks and ISDN systems*, vol. 30, no. 1, pp. 107–117, 1998.

[17] A. Heydon and M. Najork, "Mercator: A scalable, extensible web crawler," *World Wide Web*, vol. 2, no. 4, pp. 219–229, 1999.

[18] M. Najork and A. Heydon, *High-performance web crawling*. Springer, 2002.

[19] A. Z. Broder, M. Najork, and J. L. Wiener, "Efficient url caching for world wide web crawling," in *Proceedings of the 12th international conference on World Wide Web*, pp. 679–689, ACM, 2003.

[20] D. Fetterly, M. Manasse, M. Najork, and J. Wiener, "A large-scale study of the evolution of web pages," in *Proceedings of the 12th international conference on World Wide Web*, pp. 669–678, ACM, 2003.

[21] M. R. Henzinger, A. Heydon, M. Mitzenmacher, and M. Najork, "Measuring index quality using random walks on the web," *Computer Networks*, vol. 31, no. 11, pp. 1291–1303, 1999.

[22] M. R. Henzinger, A. Heydon, M. Mitzenmacher, and M. Najork, "On near-uniform url sampling," *Computer Networks*, vol. 33, no. 1, pp. 295–308, 2000.

[23] M. Najork and J. L. Wiener, "Breadth-first crawling yields high-quality pages," in *Proceedings of the 10th international conference on World Wide Web*, pp. 114–118, ACM, 2001.

[24] V. Shkapenyuk and T. Suel, "Design and implementation of a high-performance distributed web crawler," in *Data Engineering, 2002. Proceedings. 18th International Conference on*, pp. 357–368, IEEE, 2002.

[25] J. Edwards, K. McCurley, and J. Tomlin, "An adaptive model for optimizing performance of an incremental web crawler," in *Proceedings of the 10th international conference on World Wide Web*, pp. 106–113, ACM, 2001.

[26] P. Boldi, B. Codenotti, M. Santini, and S. Vigna, "Ubicrawler: A scalable fully distributed web crawler," *Software: Practice and Experience*, vol. 34, no. 8, pp. 711–726, 2004.

[27] H.-T. Lee, D. Leonard, X. Wang, and D. Loguinov, "Irlbot: scaling to 6 billion pages and beyond," *ACM Transactions on the Web*, vol. 3, no. 3, p. 8, 2009.

[28] G. Mohr, M. Stack, I. Rnitovic, D. Avery, and M. Kimpton, "Introduction to heritrix," in *4th International Web Archiving Workshop*, 2004.

[29] R. Khare, D. Cutting, K. Sitaker, and A. Rifkin, "Nutch: A flexible and scalable open-source web search engine," *Oregon State University*, vol. 1, pp. 32–32, 2004.

[30] Z. Bar-Yossef and M. Gurevich, "Random sampling from a search engine's index," *Journal of the ACM (JACM)*, vol. 55, no. 5, p. 24, 2008.

[31] C. Olston and M. Najork, "Web crawling," *Foundations and Trends in Information Retrieval*, vol. 4, no. 3, pp. 175–246, 2010.

[32] A. Broder, R. Kumar, F. Maghoul, P. Raghavan, S. Rajagopalan, R. Stata, A. Tomkins, and J. Wiener, "Graph structure in the web," in *Proceedings of the 9th International World Wide Web Conference*, 2000.

[33] A. Ntoulas, J. Cho, and C. Olston, "What's new on the web?: the evolution of the web from a search engine perspective," in *Proceedings of the 13th international conference on World Wide Web*, pp. 1–12, ACM, 2004.

[34] A. Dasgupta, A. Ghosh, R. Kumar, C. Olston, S. Pandey, and A. Tomkins, "The discoverability of the web," in *Proceedings of the 16th International Conference on World Wide Web*, pp. 421–430, 2007.

[35] S. Zheng, P. Dmitriev, and C. L. Giles, "Graph-based seed selection for web-scale crawlers," in *Proceedings of the 18th ACM conference on Information and knowledge management*, pp. 1967–1970, ACM, 2009.

[36] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," *Commun. ACM*, vol. 51, pp. 107–113, Jan. 2008.

[37] H. Karloff, S. Suri, and S. Vassilvitskii, "A model of computation for mapreduce," in *Proceedings of the Twenty-first Annual ACM-SIAM Symposium on Discrete Algorithms*, pp. 938–948, Society for Industrial and Applied Mathematics, 2010.

[38] L. Page, S. Brin, R. Motwani, and T. Winograd, "The pagerank citation ranking: Bringing order to the web.," tech. rep., 1999.