Scalable, Generic, and Adaptive Systems for Focused Crawling

Georges Gouriten
Institut Mines-Télécom
Télécom ParisTech; CNRS LTCI
75634 Parix Cedex 13, France
gouriten@telecom-paristech.fr

Silviu Maniu
Department of Computer Science
University of Hong Kong
Pokfulam Road, Hong Kong
smaniu@cs.hku.hk

Pierre Senellart
Institut Mines-Télécom
Télécom ParisTech; CNRS LTCI
75634 Parix Cedex 13, France
senellart@telecom-paristech.fr

ABSTRACT

Focused crawling is the process of exploring a graph iteratively, focusing on parts of the graph relevant to a given topic. It occurs in many situations such as a company collecting data on competition, a journalist surfing the Web to investigate a political scandal, or an archivist recording the activity of influential Twitter users during a presidential election. In all these applications, users explore a graph (e.g., the Web or a social network), nodes are discovered one by one, the total number of exploration steps is constrained, some nodes are more valuable than others, and the objective is to maximize the total value of the crawled subgraph.

In this article, we introduce scalable, generic, and adaptive systems for focused crawling. Our first effort is to define an abstraction of focused crawling applicable to a large domain of real-world scenarios. We then propose a generic algorithm, which allows us to identify and optimize the relevant subsystems. We prove the intractability of finding an optimal exploration, even when all the information is available.

Taking this intractability into account, we investigate how the crawler can be steered in several experimental graphs. We show the good performance of a greedy strategy and the importance of being able to run at each step a new estimation of the crawling frontier. We then discuss this estimation through heuristics, self-trained regression, and multi-armed bandits. Finally, we investigate their scalability and efficiency in different real-world scenarios and by comparing with state-of-the-art systems.

Categories and Subject Descriptors

H.3.7 [**Digital Libraries**]: Collection

Keywords

focused crawling, graph exploration, multi-armed bandits

1. INTRODUCTION

Thanks to better interfaces, better hardware, and the Internet, digital resources are omnipresent. People and institutions produce and consume more and more of them. Being able to efficiently collect data thus becomes increasingly important. However, the challenge is not to simply collect any data but only relevant data. An electronic music publishing company, for instance, will be particularly interested in blog posts about electronic music and probably less in those about classic painting.

Many focused data collection operations can be formalized as a topic-driven exploration of a graph. Searching for Web content is performed by following hyperlinks in the Web graph. Looking for a file in a decentralized file sharing peer-to-peer network is done by exploring the graph of the connected peers. Influential users in an online social network are found by exploring its graph. Even some social tasks such as looking for job opportunities may amount to an iterative and recursive exploration of one's connections. In addition, in all these cases, those explorations are guided by a specific need: they are topic-driven.

Accessing data in these scenarios is not free. The cost of access can be in terms of bandwidth, processing power, available time, server policies, etc. One particularly relevant example is crawling Twitter. Twitter stores vast amounts of data but only allows a comparatively tiny number of requests in a 15-minute period on its API [24]. Furthermore, not only big companies with vast IT resources need to collect data, but also individuals or small companies with more modest means.

For all those reasons, being able to build better systems for focused crawling is crucial. This is the essence of the work presented in this article. In this direction, we propose a simple and widely applicable abstraction for focused crawling, practical considerations, various new systems adaptable to different use cases, and experiments on real-world datasets.

More specifically, we offer the following contributions:

- A generic model for focused crawling, as an optimization problem on a graph, with various examples of its application; [Section 2];
- A proof that it is intractable to compute optimal crawl sequences [Section 3];
- A flexible high-level algorithm to perform focused crawling [Section 4];
- A consistent experimental framework, based on real and large datasets[1] [Sections 2 and 5];
- Pragmatic considerations on how to drive the crawl [Section 6];

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

HT'14, September 1–4, 2014, Santiago, Chile.

Copyright is held by the owner/author(s). Publication rights licensed to ACM.

ACM 978-1-4503-2954-5/14/09 ...$15.00.
http://dx.doi.org/10.1145/2631775.2631795.

[1] All datasets and source code used in the experiments are available online at `http://netiru.fr/research/14fc/index`.

- A variety of techniques to estimate the value of unknown nodes, some inspired by the literature, some conceptually novel [Section 7];
- A detailed study of the behavior of these estimators in a variety of situations [Section 8].

We summarize the related work in Section 9, just before concluding. A preliminary version of this work was presented in a national conference without formal proceedings [14]; additions with respect to that version include estimators based on the second-level neighborhood, multi-armed bandit approaches, and final experiments.

2. MODEL AND USE CASES

In this section, we define focused crawling with a generic model, explain how it applies to real-world scenarios, then introduce our experimental datasets.

2.1 Generic Model for Focused Crawling

We fix a crawling space G to be a directed graph, $G = (V, E)$. The nodes, V, are the resources that we want to crawl (e.g., Web pages). The edges, E, are the links between those resources (e.g., hyperlinks).

We assume there are non-negative *weights* (or *scores*) on both edges and nodes. The weight of a node is its relevance to the topic (e.g., how much the Web page is related to the topic of the crawl). The weight of an edge is an *a priori* indication of the relevance of the node it links to (e.g., how much the hyperlink occurs in a context that is related to the topic of the crawl). In actual crawling scenarios, the weight of a node will only be known once this node has been crawled, while the weight of an edge will be known once its source has been crawled.

These weights are computed by functions. They can be any scoring technique. For instance, in the case of the Web, we may define the score of a node, given a term, to be the tf-idf of the Web page for this term. However, we stress the generality of our approach to any *edge scoring function*, noted $\alpha : E \to \mathbb{Q}^+$, and *node scoring function* $\beta : V \to \mathbb{Q}^+$. Weights are assumed to be rationals to be representable in machine format. For the crawling space, we assume α and β to be fixed.

From the score of an individual node, we define the score of a set of nodes. Let X be a subset of V, then by definition $\beta(X) := \sum_{v \in X} \beta(v)$. Other ways of aggregating node scores could be used, we chose the sum as it is intuitive, simple, and enough for most use cases. Any other monotonically increasing scoring function could be used; the sum has the advantage of simplicity.

We now introduce some concepts specific to the setting of graph crawling. For a subset V' of V, the *frontier* of V' is the set of nodes not in V' it directly connects to. Formally:

DEFINITION 1. *We define the* frontier *of any $V' \subseteq V$ as:*

$$\text{Frontier}\left(V'\right) := \left\{ v \in V \setminus V' \mid \exists v' \in V', (v', v) \in E \right\}.$$

A crawl has a starting point, a set of nodes (e.g., some handpicked Web pages) the *seed set* which is a subset of V.

The *crawled set* is originally V_0. A *step* of the crawl consists in downloading a resource from the frontier of the crawled set, and adding it to the latter. A crawl has a limited number of steps, depending on the request rate, the bandwidth, or the hardware. We detail some typical use cases in the next part. This limit is the *crawl budget*, that we simply define to be a fixed non-negative integer n.

A *crawl sequence* is a sequence of n nodes, where each node is at the frontier of the crawled set:

DEFINITION 2. *A* crawl sequence $(v_1 \dots v_n) \in V^n$ *is any sequence of n nodes such that*

$$\forall 1 \leqslant i \leqslant n, \quad v_i \in \text{Frontier}\left(V_0 \cup \{v_1 \dots v_{i-1}\}\right).$$

The graph, the scoring functions, the seed set, and the budget define a *crawl configuration*. A *crawling system* or *crawler* returns a crawl sequence given a crawl configuration.

A crawl happens *online*, the crawler only has access to the information of the nodes of the current crawled set; their score, their outgoing edges, and their edges' scores. The scores of the nodes at the frontier are not known. We define the *performance* of a crawler as the aggregated score of the crawl sequence it returns.

For each crawl configuration, there is a fixed number of possible crawl sequences. The *optimal crawl sequences* are the crawl sequences with the highest score:

DEFINITION 3. *The set of* optimal crawl sequences *is defined as:*

$$\underset{(v_1 \dots v_n) \text{ crawl sequence}}{\arg\max} \quad \beta\left(\{v_1, \dots, v_n\}\right)$$

($\arg\max$ *returns a set as there may be different crawl sequences with the same score.)*

An *optimal crawler* is one that always returns an optimal crawl sequence. The unique β value of all the optimal crawl sequences is the *optimal score*. Edge weights are not part of the definition of optimality but, as we will see later, can be used to estimate the weights of the nodes at the frontier.

2.2 Example Use Cases

Our model covers different use cases, beyond classical focused Web crawling. The main ingredients V, E, α, β, V_0, and n can be instantiated for a variety of problems. We propose here examples of α and β, but other scoring techniques could be used in the same settings. To calculate n in the following use cases, we consider a crawl that lasts one week. For instance, in a Web crawling scenario, with one request per domain per second, it is possible to perform a total of $60 \times 60 \times 24 \times 7 \approx 6 \times 10^5$ crawling steps per domain.

Focused Web crawling [6]. This is the classical focused crawling scenario where the objective is to crawl the Web. We assume given a keyword query, used to focus the crawl.
V**:** Web pages;
E**:** hyperlinks;
α**:** tf–idf of the anchor text, w.r.t. the query;
β**:** tf–idf of the page, w.r.t. the query;
V_0**:** manually selected Web pages;
n**:** 6×10^5 requests for a unique domain, 6×10^8 for a thousand domains crawled in parallel.

In more elaborate settings [20], α and β are computed by automatically trained classifiers – this still fits within our model, with slight modification to the definition of these two functions.

Topic-centered Twitter user crawl [15]. The aim is to crawl Twitter users, retrieving their tweets from the API, and adding to the frontier the Twitter users they mention in them. We also assume a keyword query is given.
V**:** Twitter users;
E**:** mentioning relations, $(u, v) \in E$ if at least one tweet of user u mentions v;
α**:** tf–idf of all the tweets of u mentioning v w.r.t. the query;
β**:** tf–idf of all the tweets of u w.r.t. the query;
V_0**:** users obtained using the Twitter Search API;
n**:** 2×10^5 `statuses/user_timeline` requests [24].

Deep Web siphoning through a keyword search interface [3]. The goal is to siphon an entire database which is only accessible beyond an HTML form. Keyword queries are used through the form and new keywords are discovered from the pages given in response to the query.

V: keywords;

E: keyword relations, $(u,v) \in E$ if the keyword v appears in the response page obtained by submitting the form with the keyword u;

α: number of occurrences of the keyword v in the result pages for the keyword u;

β: number of records returned for a keyword;

V_0: initial small dictionary of keywords;

n: 6×10^5 requests for one domain.

In the examples so far, the crawling entity is centralized, but we can also consider cases where multiple entities perform a distributed crawl.

Gossiping peer-to-peer search [2]. One peer (e.g., in a file sharing network) issues a query and this query is propagated through gossiping to the neighboring peers.

V: peers;

E: peer-to-peer overlay network;

α: relevance, to the query, of the cached information about a remote peer;

β: relevance of a peer's data to the query;

V_0: peer issuing the query;

n: 10^4 propagation steps, limited to prevent flooding, e.g., one tenth of the total number of nodes in the network.

Using a real-world social network to answer a query [23, 11]. This example is a well-known sociological experiment where individuals are asked to use their direct social network to, e.g., forward a message to another person of the network that they are not directly connected to.

V: individuals;

E: acquaintance network;

α: assessment of an individual of her acquaintance's ability to forward the message to the right person;

β: 1 if the individual is the receiver, 0 otherwise;

V_0: user the query starts from;

n: 10^3 requests made from an individual to another, the collective effort allowed (how many people contribute).

These examples are from very different settings: the resources can be Web pages, users, machines; in some cases, a centralized entity governs the crawl, in others the process is distributed; the budget can be set to prevent flooding, or as a consequence of a time limit; etc. Yet, they all can be seen as instances of our general problem of finding the best crawl sequences starting from a given set of nodes.

We advocate that our general framework, and the algorithms we present hereafter, can be used in a wide range of scenarios, listed previously and beyond, as a basis to build efficient adaptive systems. Obviously, specific settings may also require specific adjustments. In particular, we chose to study centralized crawls. The question of managing distributed focused crawls is left for future work.

2.3 Experimental Use Cases

We chose five large and diverse datasets[2] from scenarios that correspond to actual needs.

[2]Available at `http://netiru.fr/research/14fc/index`.

Wikipedia datasets: bretagne and france. Wikipedia has a density of links and a diversity of content that makes it a good candidate to simulate a focused crawl on the Web, a very common use case.

We used scoring functions based on one keyword, *bretagne* in one dataset, and *france* in the other. The two keywords we sufficient in obtaining datasets that have realistic score distributions, and different crawling challenges, on one hand a very specialized topic, on the other a more generic one. We used the content of the French Wikipedia[3].

Let first f be a logarithm smoothening function $f : x \mapsto \log(1+x)$ that maps non-negative numbers to non-negative numbers. If x is the number of occurrences of the keyword in a Wikipedia article u, we set $\beta(u) := f(x)$. Similarly, if y is the number of occurrences of the keyword in a 100-character window around a hyperlink from u to v, we set $\alpha(u,v) := f(y)$.

Twitter datasets: happy, jazz, and weird. Crawling social networks is of particular interest given the popularity of those platforms, and their graphs are structurally different from the graph of the Web.

We built three datasets simulating a user crawl on Twitter, one of the most popular social network. Here also, we used scoring functions based on one keyword to obtain realistic score distributions with three different degrees of specialization. The three keywords are *happy*, *jazz*, and *weird*. We used the SNAP Twitter dataset [25][4].

For x the number of occurrences of the keyword in the tweets of a user u, we define $\beta(u) := f(x)$. For y the number of occurrences of the keyword in the tweets of user u mentioning user v, we let $\alpha(u,v) := f(y)$.

Dataset	**Nodes** (million)	**Non-zero nodes** (%)	**Edges** (million)	**Non-zero edges** (%)
BRETAGNE	2.2	2.0	35.6	0.5
FRANCE	″	19.2	″	6.8
HAPPY	16.9	11.0	78.0	2.4
JAZZ	″	0.6	″	0.1
WEIRD	″	3.2	″	0.4

Table 1: Size of the experimental datasets

Experimental diversity. Our datasets correspond to popular use cases, and as shown in Table 1, they cover different graph settings, in terms of size and score distribution.

3. INTRACTABLE OPTIMAL CRAWLS

In Section 2, we defined the notion of optimality for a crawling system, which depends on optimal crawl sequences. In our effort to build efficient crawling systems, it is important to investigate these optimal crawl sequences. Unfortunately, even in an *offline setting*, a setting where it is possible to access the whole graph, to determine the optimal sequence is NP-hard.

[3]February 2013 dump downloaded from `http://dumps.wikimedia.org/backup-index.html`.

[4]Formerly available at `http://snap.stanford.edu/data/twitter7.html`, it contains an estimated 30 % of all the tweets published between June and December 2009, 476 million tweets. The dataset unfortunately had to be pulled out from this Web site by request of Twitter.

PROPOSITION 1. *Let G be a graph, β a PTIME-computable node scoring function, V_0 a subset of nodes of G, and n a budget. To determine if there is a crawl sequence of score greater than or equal to a given rational r for G, β, V_0, and n is an NP-complete problem. NP-hardness holds even if V_0 is a singleton and r is an integer.*

PROOF. Membership in NP is straightforward. We guess a node sequence of size n, check that it is a valid crawl sequence (easily done in PTIME), compute its score (feasible in PTIME by hypothesis), and compare it to r.

For NP-hardness, we exhibit a PTIME many-one reduction from the LST-Graph problem described in [17]. The edge-uniform LST-Graph problem is defined as follows: given two positive integers L and W, and a directed graph $G = (V,E)$ where each edge (u,v) is annotated with a nonnegative integer *weight* $w(u,v)$, does there exist a subtree T of G such that the number of edges in T is less than or equal to L and that the sum of edge weights is greater than or equal to W? Theorem 5 of [17] shows this problem is NP-hard, by reduction from Set-Cover.

First, observe that instead of weights w on edges and budgets L and W on total edge counts and total edge weights, we can as well have weights w on nodes and budgets L' on node counts and W on the sum of node weights. To reduce the former problem to the latter, just add nodes in the middle of each edge, with weight being the weight of the edge, set weight 0 to all nodes originally in the graph, and set budget on node count L' to $2L+1$ (in a tree, there is one more node than the number of edges, and there are twice more edges now). It is straightforward to show the reduction from the original edge-uniform LST-Graph problem to this modified LST-Graph* problem, with weights on nodes and budgets on node counts and sum of node weights.

Let (L,W,G,w) be an instance of LST-Graph*. Without loss of generality, we can assume L to be at most $|V|$ (otherwise, just set L to $|V|$, the problem will have the same answer) and the graph to have at least 2 nodes (otherwise, just add one more node with no edges). Let G' be the graph obtained from $G = (V,E)$ by adding:

- an additional node r;
- for each node u of G, $L+1$ new nodes $u_1,\dots,u_{L+1}$;
- for each node u of G, a chain of $L+2$ edges (r,u_1), (u_1,u_2), $\dots$, (u_L,u_{L+1}), (u_{L+1},u).

Since L is at most $|V|$, this construction is in $O(|V|^2)$. The score of a node is their weight in the old graph, new nodes having score 0.

We claim that LST-Graph*(L,W,G,w) has a solution if and only if $(G',w,\{r\},2L+1)$ admits a crawling sequence of score greater than or equal to W. This reduction is obviously polynomial-time. We shall prove both directions of the equivalence.

First, assume (L,W,G,w) is a "yes" instance of LST-Graph*. Let T be a subtree of G of total weight at least W and of size l at most L. Let $(v^1 \dots v^l)$ be a topological sort of tree T (i.e., an ordering such that v^j descendant of v^i implies $i \leqslant j$). We consider the sequence $(v^1_1,\dots,v^1_{L+1},v^1,\dots,v^l)$ of length $L+1+l \leqslant 2L+1$; this is a valid crawl sequence starting from $\{r\}$. We complete this crawl sequence into a crawl sequence S of length exactly $2L+1$ by adding $L-l$ additional nodes $u_1,u_2,\dots,u_{L-l}$ for an arbitrary node $u \in V$ distinct of v^1. The score of S is the summed weight of T, and is thus $\geqslant W$.

Conversely, assume $(G',w,\{r\},2L+1)$ admits a crawling sequence S of score greater than or equal to W. This crawling sequence S, together with r, naturally defines a tree T' in G': the root of this tree is r; for every node v in S there is an edge from u to v where u is first node u in the sequence (r,S) such that $(u,v) \in G'$. Consider the forest $F = T' \cap V$, the restriction of T' to the nodes of G. F is not empty since S has non-zero score. F is a forest of G of length at most L (because since F is not empty, T must include at least one chain $v_1,\dots,v_{L+1},v)$ and of summed weight greater than or equal to W (because new nodes do not contribute to weight). We just have to show that F is connected. Since T is connected, F is disconnected only if there are two chains $u_1,\dots,u_{L+1},u$ and $v_1,\dots,v_{L+1},v$ in T with $u \neq v$. But the length of these two chains combined is $2L+2 > 2L+1$, which is impossible to fit inside T. □

An immediate corollary of Proposition 1 is that it is unfeasible for practical purposes to determine whether a specific crawl sequence is close enough in score to the optimal. Consequently, it is also intractable to find an optimal crawl sequence. As in [17], we leave as an open problem the possibility of approximating the optimal score or finding a crawling sequence with a score being a factor of the optimal one.

In order to be optimal, a crawler would have to produce an optimal crawling sequence – the result above shows that this is NP-hard. This implies that even a crawler using a greedy strategy with an oracle estimator (concepts that we introduce later) cannot be optimal in the general case.

On a side note, if we do not use the sum of individual node weights but the *count* of nodes having non-zero weight as subset scoring, a greedy solution *does* work as long as there are some non-zero nodes in the frontier. The proof is straightforward, as adding one non-zero node always adds one to the total score, which is the best that can be done at any given point.

4. HIGH-LEVEL ALGORITHM

We gave in Section 2 the intuition behind a crawling process. In this part, we clarify this introducting a generic algorithm and the subsystems that we will investigate later: `scoreFrontier`, `getBatchSize`, and `getNextNodes`.

Algorithm 1: High-level algorithm

```
input  : seed subgraph G_0, budget n
output : crawl sequence V, with a score as high as possible
1  V ← ();
2  G' ← G_0;
3  budgetLeft ← n;
4  while budgetLeft > 0 do
5  |  frontier ← extractFrontier(G');
6  |  scoredFrontier ← scoreFrontier(G', frontier);
7  |  b ← getBatchSize();
8  |  NodeSequence ← getNextNodes(scoredFrontier, b);
9  |  V ← (V, NodeSequence);
10 |  for u in NodeSequence do
11 |  |  G' ← G' ∪ crawlNode(u);
12 |  budgetLeft = budgetLeft − b
13 return V
```

Algorithm 1 maintains a crawled graph G'. This is the union of the subgraph induced by the crawled set, and the set of its outgoing edges, pointing to the frontier. G' is initialized to G_0, the subgraph of G induced by V_0 (line 2), it is then updated as more nodes are crawled (lines 10–11). The main loop of the crawl (lines 4–12) iterates as long as there is some budget left. The method `getBatchSize` allows to create a *batch* of crawling steps of size b, which avoids estimating the frontier between each crawled node – instead, a batch of b nodes are crawled using the current estimation of the scored frontier.

The method `extractFrontier` extracts the frontier from the crawled graph (line 5). As we explained in Section 2, the crawler does not have access to the scores of the nodes at the frontier, so `scoreFrontier` assigns a score *estimation* to these nodes. For that

purpose, it has access to the crawled graph, which can serve as a training set. `getNextNodes` decides which should be the b nodes to crawl, depending on the scored frontier. The crawl sequence V, the crawled graph, and the budget are then updated accordingly (line 9–12), `crawlNode` consisting in retrieving a node, its actual score, and its outgoing edges, which are added to G'.

The above pseudocode gives a high-level overview of a generic focused crawler, but the details might change slightly in practical implementations, for instance to implement incremental statistical learning models.

5. EXPERIMENTAL FRAMEWORK

Now that we have a good understanding of the general context, we will study different systems in detail and present several empirical findings. It is therefore important to understand the experimental framework we used in order to ensure the consistent quality of these findings.

Baseline. For any crawl configuration with a meaningful budget, it is not possible to find the optimal crawl sequence. We thus could not compare our crawlers to an absolute optimal baseline. However, we specified in each experiment a baseline crawler, and compared the different crawlers, including those from the literature, relatively.

Testing configuration. We also tested our crawlers with enough configurations to ensure statistical significance. We tested fifty different configurations. For each experimental dataset (introduced in Section 2.3), we built ten seed sets, made of fifty different nodes chosen uniformly at random among those having non-zero weights. We ran the crawler for each seed set to obtain a seed score. We then computed their arithmetic mean to obtain a *dataset score*. This score was divided by the dataset score of a baseline, specified in each experiment, to obtain a normalized score. We eventually computed the geometric mean of the normalized score among the different datasets to obtain a *global score*.

Crawl budget. The maximum crawl budget we used is 10^5. As explained in Section 2.2, this would be a crawl campaign of three days and a half on Twitter [24], and a bit more than one day on Wikipedia. It is reasonable both in terms of time and freshness of the crawled data.

Implementation. The running time and memory consumption of a crawler is critical. For the largest experiments, we used the C++ programming language and stored the graph and the crawler metadata in the RAM. The RAM memory cost of our datasets was important – especially for the Twitter graph. Thus, all experiments were run on a Linux PC with 48 GB RAM. The CPU model was an Intel(R) Core(TM) i7-3820 CPU @ 3.60GHz.

Each subsystem of the crawler was an abstract C++ class, with derived classes for different possible implementations (greedy vs altered greedy, different estimators), that provide a number of virtual methods, especially `scoreFrontier` and `getNextNodes`. The source code, the datasets, and instructions to run the experiments are available online.[5]

We used the *Boost* graph library[6]. For the linear regression, we used the *dlib* C++ library[7] that implements an *incremental* version of the recursive least squares algorithm.

[5] `http://netiru.fr/research/14fc/index`
[6] `http://www.boost.org/libs/graph`
[7] `http://dlib.net/`

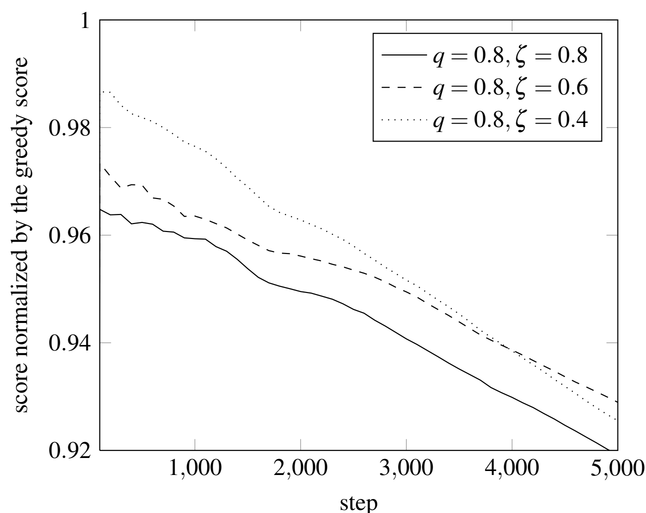

Figure 1: Best altered greedy parameters for JAZZ at step 5,000

6. STEERING THE CRAWLER

Before looking at how to build a good estimation of the frontier (`scoreFrontier`), we think here about which nodes to pick once the frontier has been estimated (`getNextNodes`), and what the impact of using batch processing is (`getBatchSize`).

To study those questions, in this section, we allow the crawlers not to worry about the estimation. They have access to the perfect estimator, the *oracle*, meaning that `scoreFrontier` returns the exact scores of the nodes at the frontier. The estimators that we use in practice are quite far from the oracle, as we will see in Section 8.1, so we consider only properties that look reasonable as invariants.

6.1 Rich People have Rich Friends

`getNextNodes` looks at the *estimated* frontier and returns the next nodes to crawl. We investigate here what this function should do, limiting ourselves to a batch size of 1, thus `getNextNodes` returns only one node. The estimated score of a node is noted $\tilde{\beta}(v)$. We also refer to `getNextNodes` as the *strategy* of the crawler.

In our online and incremental situation, a *greedy strategy* is a reasonable option:

STRATEGY 1. greedy strategy $= \underset{v \in \text{Frontier}(V')}{\arg\max} \; \tilde{\beta}(v)$

Yet, a pure greedy strategy could miss interesting parts of the graph. A node bridging to a rich subgraph with a low score would never be selected. This is the usual trade-off between exploitation and exploration. To test this risk, we consider *altered greedy* strategies, defined by a *probability* $q \in [0,1]$ and a *ratio* $\zeta \in [0,1]$.

STRATEGY 2. altered greedy $=$
with probability q, $\underset{v \in \text{Frontier}(V')}{\arg\max} \; \tilde{\beta}(v)$,
with probability $1-q$,
$\text{random}\left(\left\{ v \in \text{Frontier}(V') \mid \tilde{\beta(v)} \geqslant \tilde{\zeta} \times \max_u(\beta(u)) \right\}\right)$

We tested the altered greedy strategies for $q = 0.2, 0.4, \ldots, 1.0$ and $\zeta = 0.0, 0.2, \ldots, 0.8$.

Figure 1 illustrates well what we found with all the datasets. For any number of steps greater or equal to a few thousands, the best altered greedy strategies are the one with the highest probability and ratio, and all the randomized strategies fall behind greedy.

We explain this with the "rich people have rich friends" property. The rich nodes (with a high score) cannot, statistically, have as friends (connected nodes) only poor nodes, they also have some rich nodes. When the frontier has a significant size, crawling the rich nodes is generally enough to get to the other rich nodes.

However, for experiments below a thousand steps, we sometimes saw altered greedy beat greedy, even with risky probability and ratio. With a small frontier, some rich parts can "hide" behind poor nodes, which then become more interesting. It is a phenomenon worth mentioning, but the gain was not very important and our work is mostly on large crawls so we did not investigate further on.

Greedy being a clear winner for any crawl of significant size, we use the greedy strategy for the rest of the study, `getNextNodes` returns the nodes with the top estimated scores.

6.2 The Batch Disadvantage

As introduced in Section 4, `getBatchSize` can be used to reduce the computation time. It does so by returning an integer, b, the *batch* size, greater than 1. It is then used to crawl a set of nodes of that size, instead of a unique node, before calling `scoreFrontier`, which is a relatively costly operation.

However, increasing the batch has a performance cost. We tested crawlers differing only by their batch sizes. The estimator was the *oracle*. For each crawler, we computed a global score according to the experimental framework defined in Section 5. The baseline for the score degradation was the score of the crawler with $b = 1$. If the crawler had no estimated node in the frontier, it picked the next node randomly.

	Step			
Batch size	100	1,000	10,000	100,000
2	0.4	2.2	3.9	6.4
8	1.3	6.5	12.8	18.3
32	6.6	6.5	17.5	24.3
128	38.8	10.7	19.9	29.5
1024	38.8	74.3	25.8	35.9

Table 2: Score degradation (%) for different batch sizes

In Table 2, we see the importance of the performance degradation, even for a batch size of 2. We explain it with two main arguments. First of all, the crawler is sometimes forced to pick nodes randomly, which, on average, degrades the performances (see Section 6.1). Secondly, the sooner a rich node is crawled, the sooner its connections are added to the frontier. Adding a rich node later creates a *score delay*, that will remain during the crawl, and increase if this delay is repeated.

Even for a batch size of 2, the performance degradation is notable. For this reason, we will, for the rest of the study, look for scalable estimation techniques, allowing to refresh of the estimation at each step.

7. FRONTIER ESTIMATORS

`scoreFrontier` is our last system to study, and the most difficult. It takes as an input the crawled graph and the frontier, and returns a `scoredFrontier`, an estimation of the β value of each node of the frontier, $\widetilde{\beta}$.

Thanks to the previous parts, we now have some precise ideas on what we would like for `scoreFrontier`. We look for a scalable system able to identify the top nodes of the frontier. We also call such a system an *estimator*.

In this part, we formalize frontier estimators, some adapted from the literature, some new. We are in the context of a specific crawl, V' is the set of crawled nodes, E' the set of known edges – including edges from V' to $\text{Frontier}(V')$, and $d_\text{o} : V \to \mathbb{N}$ (resp., d_i) the number of known outgoing (resp., incoming) edges of a node. The estimators are defined by their node weight estimation function $\widetilde{\beta} : \text{Frontier}(V') \to \mathbb{Q}^+$. We defined a `short_name` for each estimator.

7.1 State-of-the-Art

We start our list looking at estimators from the literature that we adapted and translated using our model.

BFS. `bfs` is the *breadth-first search* estimator, a simple estimator based on a common crawling technique.

ESTIMATOR 1 (`bfs`). $\widetilde{\beta}(v) = \frac{1}{l(v)+1}$, *where $l(v)$ is the distance of v to V_0.*

This estimator is naive and we can expect poor results. We used it as a bottom line.

As we discuss in Section 9, most focused crawling systems from the literature mostly deal with how to compute the score of a node or an edge (i.e., how to define α and β), once they have been crawled [20]. Not many are about how to estimate those values *before* they are crawled (i.e., how to define $\widetilde{\beta}$). The exceptions are the following two approaches. They are inspired by PageRank and perform iterative computations on the crawled graph.

Navigational Rank. Navigational Rank [12] is a two-step node importance computation specifically designed for focused crawling. The first step is an iterative propagation from offsprings to ancestors, combined with the actual node score:

$$NR_1(v)^{t+1} = \eta \times \beta(v) + (1-\eta) \times avg_{(v,u)\in E'} \frac{NR_1(u)^t}{d_\text{i}(u)}$$

where $NR_1(u)^t$ is the node score at the iteration step t, and η is a parameter affecting convergence speed.

The second propagation step is performed only on the frontier nodes, and is from ancestors to offspring, as follows:

$$NR_2(v)^{t+1} \eta \times NR_1(v) + (1-\eta) \times avg_{(u,v)\in E'} \frac{NR_2(u)^t}{d_\text{o}(u)}.$$

We must make it clear that [12] contained an error in both equations (1) and (2). The senses of the propagations were reversed. It is fixed in the equations we just presented and has been confirmed with the authors of [12].

We define `nr`, the *Navigational Rank* estimator, as the NR_2 score.

ESTIMATOR 2 (`nr`). $\widetilde{\beta}(v) = NR_2(v)$.

One step of `nr` estimation requires two successive iterative computations (with perhaps a few dozens steps) on the whole crawled graph. This results in an overall quadratic complexity, with important multipliers, for each frontier estimation. As we will see in Section 8.1, this will have major consequences on the running time.

OPIC. OPIC [1] is an algorithm designed to estimate the PageRank of a node in a crawling situation.

OPIC maintains two per-node counters during the crawl: $C(v)$ – the *cash* value of of a node, initially set to 1, and $H(v)$, its cash history, starting at 0. It also keeps a global counter G, the entire cash accumulated in the system.

The estimation takes three steps:

1. the node v with the highest cash among all encountered nodes is selected (ties resolved arbitrarily), and its history is updated with the current cash value $H(v) = H(v) + C(v)$,
2. for each outgoing node u of v, the cash value is updated $C(u) = C(u) + \frac{C(v)}{d_{o(v)}}$,
3. the global counter is incremented and the cash value of v is reset, $G = G + C(v)$ and $C(v) = 0$.

Since OPIC does not take into account edge scores, we changed the second step with the following formula:

$$C(u) = C(u) + \frac{C(v)}{\sum_{(v,w)\in E'} \alpha(v,w) \times C(w)} \times \alpha(v,u) \times C(u)$$

`opic` is based on the three counters defined above.

ESTIMATOR 3 (`opic`). $\widetilde{\beta}(v) = \frac{H(v)+C(v)}{G+1}$.

7.2 Looking at the Neighborhood of a Node

The estimators from the research literature are not entirely satisfying for focused crawling. `nr` is computationally costly. `opic` is PageRank specific and initially does not take into account node and edge weights. We thus defined new estimators, starting here with intuitive heuristics.

For those heuristics, we combine in simple ways the scores of the nodes already crawled and of their outgoing edges. For a node of the frontier v, $P(v)$ is the *set of parent nodes*, or the *first-level neighborhood* of the node v. $P(v) = \{u \in V' \mid (u,v) \in E'\}$. The nodes of V' pointed to by the nodes of the first-level neighborhood constitute the *second-level neighborhood*.

The `fl_` estimators are based on the first-level neighborhood, and the `sl_` estimators on the second-level.

ESTIMATOR 4 (`fl_n fl_e fl_ne sl_n sl_e sl_ne`).

`fl_deg`: $\widetilde{\beta}(v) = d_i(v) = |P(v)|$

`fl_n`: $\widetilde{\beta}(v) = \sum_{u\in P(v)} \beta(u)$

`fl_e`: $\widetilde{\beta}(v) = \sum_{u\in P(v)} \alpha(u,v)$

`fl_ne`: $\widetilde{\beta}(v) = \sum_{u\in P(v)} \beta(u)\alpha(u,v)$

`sl_n`: $\widetilde{\beta}(v) = \sum_{u\in P(v)} \sum_{\substack{w\in V' \\ u\in P(w)}} \beta(w)$

`sl_e`: $\widetilde{\beta}(v) = \sum_{u\in P(v)} \sum_{\substack{w\in V' \\ u\in P(w)}} \alpha(u,w)$

`sl_ne`: $\widetilde{\beta}(v) = \sum_{u\in P(v)} \sum_{\substack{w\in V' \\ u\in P(w)}} \beta(w)\alpha(u,w)$

We will see how those heuristics perform later in the article. Before that, we looked at statistical correlations between the heuristics and the node scores. We used the Pearson correlation coefficients for this purpose. We obtained the results in a setting where the full graph is known so we have to look at those measures cautiously. However the intuitions we gain from this analysis proved, as we will see, robust enough for the online setting. We averaged the coefficients geometrically over the five graphs. Lastly, we looked also tried to smooth the values with the function $f : x \mapsto \log(1+x)$ (*log* row in the following table).

Type	`fl_deg`	`fl_n`	`fl_e`	`fl_ne`	`sl_n`	`sl_e`	`sl_ne`
orig.	0.063	0.127	0.073	0.090	0.170	0.199	0.203
log	0.269	0.373	0.402	0.412	0.273	0.360	0.356

Table 3: Pearson correlation coefficients

From Table 3, we learn several things. First of all, the correlation between any of the features and the actual node score is positive. Secondly, logarithmic smoothening reinforces significantly the correlations (on a relative scale). We thus decided to use the smoothened version of those estimators in the rest of the study ($\widetilde{\beta}(v) = f(\widetilde{\beta}(v))$. Lastly, we observe some differences between the heuristics, but there is no clear winner as the coefficient remains small.

7.3 Linear Regression

These positive correlations gave us the intuition to use these estimators as features for a linear regression. We define `lr_fl` and `lr_sl`, respectively the *first-level* and *second-level linear regression* estimators.

These estimators are linear combinations for which the coefficients are trained. The crawled graph is used as the training data and the coefficients are updated before each frontier estimation. We used the *least-squares linear regression* with incremental solvers. This allowed us to not fully retrain the models for each estimation.

ESTIMATOR 5 (`lr_fl lr_sl`).

`lr_fl`: $\widetilde{\beta}(v)$ = *trained linear combination of the* `fl_` *estimators.*

`lr_sl`: $\widetilde{\beta}(v)$ = *trained linear combination of the* `fl_` *and* `sl_` *estimators.*

To have a first intuition on those estimators, we performed a R^2 analysis. It is a measure of the precision we can expect from the linear models, but shown here with full graph knowledge.

Type	`lr_fl`	`lr_sl`
orig.	0.030	0.075
log	0.221	0.230

Table 4: Linear regression fit (R^2), geometrically averaged over the five graphs

From Table 4, we confirm the logarithmic smoothening is beneficial, as well as the addition of the second level features to the training, which means we do not have an overfitting problem. However, we also see that, even if the R^2 fit is better for `lr_sl`, its value remains low in the absolute.

7.4 Reinforcement Learning

As we saw in the previous statistical analysis and as we will see more in details in Section 8.1, the intuition gained from the observation of the different estimators is that they perform differently at different stages. For instance, in some graphs, `fl_e` performs on average very well at the beginning of a crawl but poorly later. In this situation, reinforcement learning allows us to pick the right model at the right time.

Initial multi-armed bandit strategy. We chose to model our situation as a multi-armed bandit problem [22]. There is a room full of casino machines (the bandits) with different reward distributions. A player enters the room with a budget of n lever pulls and looks for a strategy to maximize its total reward. In our case, a slot machine is an estimator, and a reward is the weight of the top node returned. Playing a lever is the action of crawling the top node of an estimation model.

In this situation, the challenge is to balance properly the exploration (the player wants to test as many slot machines as possible) and the exploitation (if the player has found the best machine, he should focus on this one). A first usual way is to use an *epsilon-greedy* bandit strategy. Let $\varepsilon \in [0,1]$ be the *epsilon-greedy parameter*. With a probability ε the slot machine with the highest average

reward is used. With a probability $1-\varepsilon$, the slot machine used is chosen uniformly randomly.

ESTIMATOR 6 (mab_ε). $\widetilde{\beta}(v)$ = *output of an epsilon-greedy strategy*

With $\varepsilon = 0.1$, this estimator already gives interesting results. However, it has two major potential shortcomings. It gives the same importance to old and to new rewards, when we might want to favor new information. It also does not adapt well to variations of pace in the average reward changes. We want the exploitation–exploration ratio to vary with how dynamic the context is. We propose a new simple and robust solution to solve those two issues.

ε-first with variable reset strategy. Let $r \in \mathbb{N}$ be the *reset parameter*, a *bandit strategy with reset* is so that, every r steps, all the slot machines average rewards are set to 0. From a different perspective, let n be the number of lever pulls so far, the average reward is calculated with a weight of 1 for the rewards at the steps from $max(0, n-r)$ to n, and 0 for the more ancient ones.

This notion of reset is particularly intuitive in the context of an *epsilon-first bandit strategy*. Let $\varepsilon \in [0,1]$ be the *epsilon-first parameter* and N the total amount of lever pulls. For the steps between 0 and $\lfloor \varepsilon \times N \rfloor$, the slot machine used is chosen uniformly randomly. For the rest of the steps, the slot machine with the highest average reward is used.

ESTIMATOR 7 (mab_ε-first). $\widetilde{\beta}(v)$ = *output of an epsilon-first strategy*

The *epsilon-first with reset bandit strategy* is a succession of $\lfloor \frac{N_{total}}{r} \rfloor + 1$ epsilon-first bandit strategies, all of them except the last one with $N = r$. The last one is so that $N = N_{total} - r \times \lfloor \frac{N_{total}}{r} \rfloor$.

Let $f : \mathbb{N}^2 \rightarrow \mathbb{N}$ be the *reset variation function*, p the number of times the best estimator at the end of the exploration phase has been the same, an *epsilon-first bandit strategy with variable reset* is an epsilon-first bandit strategy with reset where $r = h(p)$. Obviously, we will pick h increasing with respect to p.

ESTIMATOR 8 (mab_var). $\widetilde{\beta}(v)$ = *output of an epsilon-first with variable reset strategy*

8. COMPARING THE ESTIMATORS

We tested the quality of the different estimators in terms of precision, running times, and ability to lead a crawl. We chose to use a batch size of 1 and `getNextNodes` returning the top nodes (the greedy strategy from Section 6.1), due to the reasons explained in Section 6. The crawl is thus driven by the estimated top node.

8.1 With the Same Crawl Sequence

In this section, we force the crawlers to have the same crawl sequence, the crawl sequence returned by the crawler with the oracle as its estimator.

Running times. We look here at how long an estimator takes to compute its estimation. This running-time depends on the size of the estimated frontier and the scores of the crawled graph. Crawl sequences thus determine running times. That is why we chose to compare the crawlers with the same crawl sequence.

As explained in Section 2.2, the different use cases require different running-times. However, for the less demanding use cases, there is 1 second between two requests for a unique Web domain, and 3 seconds for the Twitter API. The estimators must at least scale to those examples.

We measured how long the frontier estimation took at different steps on a Wikipedia (FRANCE) and a Twitter (HAPPY) graph. We chose only one graph from Wikipedia and one from Twitter as the main variable affecting the running time is the topology of the graph, rather than the scores themselves.

Dataset	Evaluator	100	1,000	10,000	100,000
FRANCE	nr	2,832.1	19,720.5	N/A	N/A
	opic	1.9	2.5	4.6	4.7
	ne_fl	0.2	0.1	0.1	0.1
	lr_fl	0.2	0.2	0.1	0.1
	mab_var_fl	0.6	0.3	0.2	0.2
	ne_sl	8.5	27.1	2.0	6.1
	lr_sl	8.5	27.2	2.0	6.1
HAPPY	nr	45,965.7	105,209.3	N/A	N/A
	opic	1.8	1.6	1.9	2.5
	ne_fl	0.3	0.1	0.2	2.1
	lr_fl	0.5	0.1	0.2	2.1
	mab_var_fl	1.1	0.3	0.5	3.9
	ne_sl	111.1	24.5	63.3	240.5
	lr_sl	111.4	24.5	63.3	241.0

Table 5: Running-times (in ms) of the evaluators at various steps

From Table 5, we see that the `nr` running time is several orders of magnitude above what we require. Simulating a few thousand steps of crawl took hours, compared with seconds for the others estimators. The second-level neighborhood estimators can scale, but have a non negligible cost, and might not be usable for restricted running time budgets. The other estimators have sub-millisecond running-times on both graphs, and are thus able to fit for most crawling restrictions.

Regarding `nr`, we remarked that the analysis performed in [12] is based on graphs that are much smaller than those we use here, typically graphs of small individual websites.

Precision. A precise estimation will return, as a top node, a node that is not too far in score from the oracle top node. To quantify the precision of an estimator, we measured the distance between the score of the top node it returns to the score of the oracle node. The distance being the score difference.

Following the same crawl sequence allows us to see the variations in performance at the different stages. The results were noisy so we smoothed them, arithmetically averaging the distance in a window of 1,000 steps.

We computed the precision for the estimators based on the first-level neighborhood only. The intuitions we gained from them are similar for the second-level neighborhood. We also did not look at multi-armed bandit strategies at that time since those experiments led us to them.

Figure 2 illustrates well the general properties we observed. First of all, the average distance decreases in the long run. At the beginning, there are a lot of rich nodes that can easily be missed. In the long run, the richest nodes will have been consumed. The average distance logically tends to be lower. Secondly, making abstraction of this tendency, the estimators perform differently at different steps. Here, we can see that `ne` does very well at first, but then loses ground to `n`. Lastly, on average, we observed the common trend that the simple neighborhood-based estimators seem to perform best at the start, while the linear regression estimator catches up in the later stages of the crawl. However, we could not find a global winner

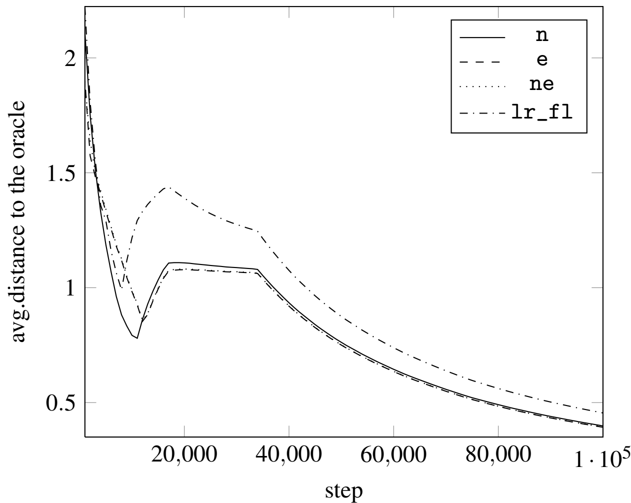

Figure 2: Estimators precision for BRETAGNE

among the different heuristics or regressions, which gave us the intuition of the multi-armed bandit strategies.

8.2 Driving the Crawl

In the previous part, we looked at situations where the crawl sequence was decided by the oracle. We investigate here what happens when the estimators are themselves leading the crawl.

Multi-armed bandit estimators. It is convenient to first study the estimators based on multi-armed bandit strategies. We experimented several of them with different parameters. For all these estimators, the slot machines are the estimators based on the first-level neighborhood, including the linear regression. The reason why we did not look at estimators based on the second-level neighborhood is explained in the next paragraph.

We used the generic experimental framework introduced in Section 5 to obtain global scores, using the oracle greedy as a baseline. Those global scores allow to gauge the overall potential of each estimator, disregarding as much as possible the particular cases.

`mab_`ε and `mab_`ε`-first` have been parametrized with $\varepsilon = 0.1$. `mab_var-0.1-1000` and `mab_var-0.2-200` have respectively for parameters $\varepsilon = 0.1, r = h(p) = 1000 \times (p+1)$ and $\varepsilon = 0.2, h(p) = 200 \times (p+1)$.

Type	100	1,000	10,000	100,000
ε	0.450	0.481	0.477	0.495
ε-first	0.409	0.501	0.484	0.490
var-0.1-1000	0.383	0.439	0.420	0.494
var-0.2-200	0.427	0.413	0.461	0.458

Table 6: Global performance of the multi-armed bandit estimators

Looking at Table 6, our first observation is that the multi-armed bandits are very comparable in terms of performances. We can also already notice that their global score is stable at different steps. As we will see below, this stability is a major difference with the other estimators. For the other experiments, in order not to overcrowd our figures, we chose to only consider `mab_var-0.2-200`.

Overall comparison. We ran crawls on the five datasets for the estimators introduced in Section 7, with a few exceptions. As explained in the previous paragraph, we only show one estimator based on multi-armed bandit, `mab_var-0.2-200`. We also excluded `nr` as its running-time was too long. Lastly, we studied the estimators based on the second-level neighborhood on a few examples, and found out that their crawl performances are not better than those of the first-level. Since their running times are too large, we decided not to compute the full results for those estimators.

Figure 3 shows the crawl performance for the different estimators in two scenarios. It illustrates interesting properties of the estimators. First of all, the breadth-first estimator, `bfs`, and the PageRank-based estimator, `opic`, are clearly worse than the other estimators. However, they remain interesting baselines. Note that we also tried the original OPIC, that does not use edge or node scores, without any improvement. The first-level neighborhood estimators, on the other hand, usually perform quite well, but with different quality in different situations. This confirms the results found in Section 8.1. Regarding the linear regression, it seems to do better at the end than at the beginning, and achieves good results at the end of the crawl. At last, the multi-armed bandit strategy seems to perform well all along the crawl, staying on the highest part.

Estimator	100	1,000	10,000	100,000
bfs	0.147	0.132	0.130	0.207
opic	0.283	0.184	0.205	0.287
n	0.358	0.280	0.362	0.467
e	0.594	0.560	0.457	0.377
ne	0.583	0.570	0.466	0.378
lr_fl	0.325	0.382	0.466	0.504
mab_var-0.2-200	0.427	0.413	0.461	0.458

Table 7: Global score of the estimators

Those results are confirmed by Table 7. It shows the global score (averaged over the five graphs), normalized with the oracle greedy. This score gives a general idea of the ability of an estimator to lead a crawl. `bfs` and `opic` are significantly worse than the rest. `n` behaves worse than the other heuristics at first, except in later stages of the crawl. `e` and `ne` are almost equivalent, leading a good crawl at the beginning but not towards the end. `lr_fl` is not great at the beginning but performs well in the long run. `mab_var` does stably well all along the crawl.

8.3 Building the Right Crawling System

We can now come up with reasonable recommendations in order to build a crawler that will perform generally well. This crawler will implement the high-level algorithm with `getNextNodes` as greedy, and `getBatchSize` as the constant 1.

To build `scoreFrontier` it is more complicated. First, we suggest to compute the estimators based on the first-level neighborhood. They are not very costly and usually helpful. From there, it is interesting to perform an incremental linear regression using those estimators as features. Eventually, combining those different estimators with a multi-armed bandit strategy should allow to pick the best estimator at different steps of the crawl.

9. RELATED WORK

General works. Most works on focused crawling addressed web page and hyperlink scoring or classification. Those works are com-

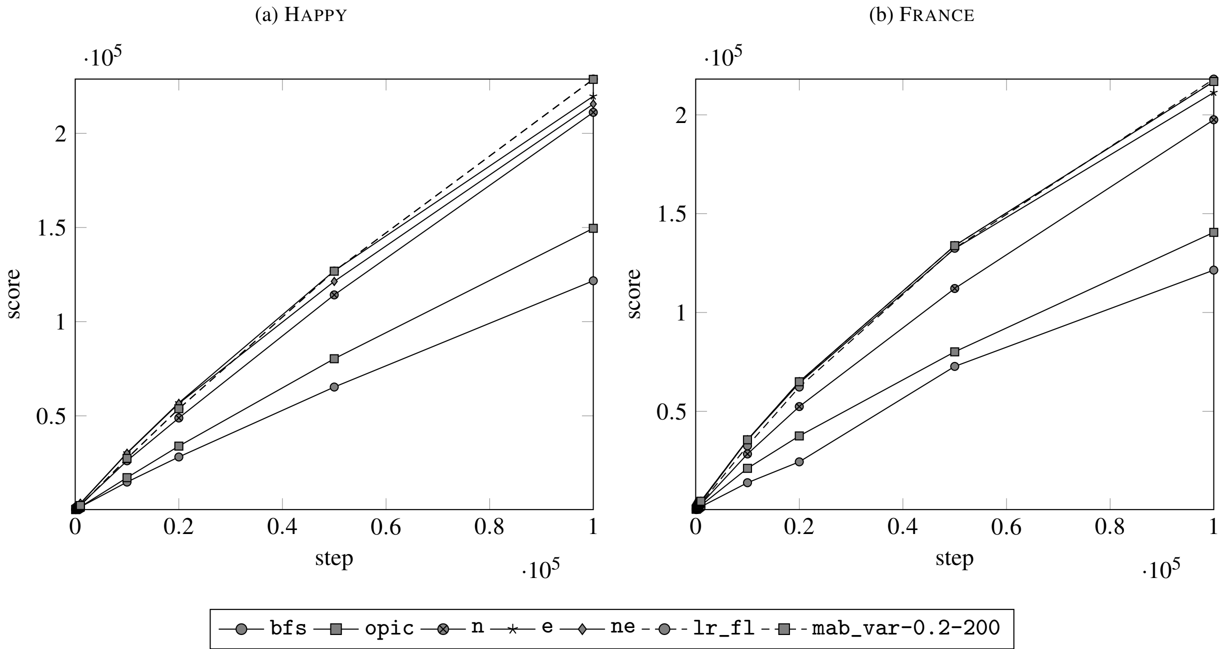

Figure 3: Average graph scores for various estimators leading the crawl of two graphs

plementary with ours, and serve as inspiration for the scoring functions α and β. A 2005 survey [20] studied the different options available at that time for focused crawling classifiers and scorers. The Fish-search [9] and Shark-search [16] algorithms score pages and links with tf-idf measures. [7, 5] put forward a crawler made of a classifier, to distinguish relevant pages, and a distiller, to identify pages more likely to point to other relevant pages. Classifiers used range from SVMs [19], HMMs [18, 4], to reinforcement learning [21]. An experimental study [19] proved that a crawler using both link and page scores outperforms single-score techniques.

Steering the crawler. The complexity result regarding finding the optimal crawl score was proved by reduction to the length-constrained maximum-sum subtree problem [17]. The altered greedy strategy was inspired by greedy randomized adaptive search procedures [13]. To the best of our knowledge, the impact of using batch processing for the frontier estimation has not been studied before.

Frontier estimation. A first idea to use some graph properties to estimate a frontier node score is formulated in [10] but only applies to Web pages, requires that the crawler is aware of backlinks to pages, and is only compared to one other system.

We compared our crawler with two main state-of-the-art systems: OPIC [1], which aims at estimating a PageRank score (on a similar idea, the RankMass crawler [8] intends to maximize coverage of high Personalized PageRank nodes); and the Navigational Rank [12], that uses a two-way propagation to estimate frontier scores and found to perform better than previous approaches.

The multi-armed bandits with variable reset was inspired by the adaptive ε-greedy exploration for reinforcement learning [22].

10. CONCLUSIONS

In this paper, we formulated the focused crawling problem as a graph exploration problem, the graph having weighted nodes and weighted edges. We then illustrated how this model can be applied to different use cases. This formalization allowed us to introduce a generic algorithm to perform focused crawling and to identify several important subproblems. From there, we studied those subproblems one by one. We demonstrated the NP-hardness of the offline optimal crawl problem, the "rich people have rich friends" property, and the batch disadvantage. Then, we looked at different techniques to estimate the frontier. We adapted state-of-the-art estimators to our formalization and proposed various new estimators. Finally, we dissected them and evaluated their running-times, their precision in identifying the best nodes, and their performance when leading a crawl.

Our formalization of focused crawling is novel and high-level. The results we present can be applied in many cases. The systems we propose have very good performance. We believe this is a significant contribution towards scalable, generic, and adaptive systems for focused crawling.

This work also allowed us to identify promising opportunities. As we proved the NP-hardness of the offline optimal crawl problem, finding an approximation technique is an interesting new challenge. The question of a distributed focused crawling systems is open and exciting. We also did not address the issue of refreshing a node already crawled, it should be possible to reuse our model to integrate it. Finally, a more thorough study on the multi-armed bandits techniques is worthy of investigation.

Acknowledgments

We are grateful to Bogdan Cautis for his valuable inputs. We also thank the Futur et Ruptures program of Institut Mines–Télécom, the France – Hong Kong PHC Procore 2012 grant 26893QA, as well as the French government under the X-Data project, for their financial support.

11. REFERENCES

[1] Serge Abiteboul, Mihai Preda, and Gregory Cobena. Adaptive on-line page importance computation. In *WWW*, 2003.

[2] André Allavena, Alan J. Demers, and John E. Hopcroft. Correctness of a gossip based membership protocol. In *PODC*, pages 292–301, 2005.

[3] Luciano Barbosa and Juliana Freire. Siphoning hidden-web data through keyword-based interfaces. *JIDM*, 1(1):133–144, 2010.

[4] Sotiris Batsakis, Euripides G M Petrakis, and Evangelos Milios. Improving the performance of focused web crawlers. *Data & Knowledge Engineering*, 68(10):1001–1013, 2009.

[5] Soumen Chakrabarti, Kunal Punera, and Mallela Subramanyam. Accelerated focused crawling through online relevance feedback. In *WWW*, pages 148–159, 2002.

[6] Soumen Chakrabarti, Martin van den Berg, and Byron Dom. Focused crawling: a new approach to topic-specific web resource discovery. *Computer Networks*, 1999.

[7] Soumen Chakrabarti, Martin van den Berg, and Byron Dom. Focused crawling: a new approach to topic-specific Web resource discovery. *Computer Networks*, 31(11-16):1623–1640, 1999.

[8] Junghoo Cho and Uri Schonfeld. RankMass Crawler: a Crawler with High Personalized PageRank Coverage Guarantee. In *VLDB*, pages 375–386, may 2007.

[9] Paul de Bra, Geert-Jan Houben, Yoram Kornatzky, and Reinier Post. Information Retrieval in Distributed Hypertexts. In *RIAO*, 1994.

[10] Michelangelo Diligenti, Frans Coetzee, Steve Lawrence, C. Lee Giles, and Marco Gori. Focused Crawling Using Context Graphs. In *VLDB*, pages 527–534, 2000.

[11] Peter Sheridan Dodds, Roby Muhamad, and Duncan J. Watts. An experimental study of search in global social networks. *Science*, 301(5634):827–829, August 2003.

[12] Shicong Feng, Li Zhang, Yuhong Xiong, and Conglei Yao. Focused crawling using navigational rank. In *CIKM*, pages 1513–1516, 2010.

[13] Thomas A. Feo and Mauricio G.C. Resende. Greedy randomized adaptive search procedures. *Journal of Global Optimization*, 6:109–133, 1995.

[14] Georges Gouriten, Silviu Maniu, and Pierre Senellart. Exploration adaptative de graphes sous contrainte de budget. In *BDA*, 2013.

[15] Georges Gouriten and Pierre Senellart. API Blender: A uniform interface to social platform APIs. In *WWW*, April 2012. Developer track.

[16] Michael Hersovici, Michal Jacovi, Yoelle S. Maarek, Dan Pelleg, Menanchem Shtalhaim, and Sigalit Ur. The Shark-Search algorithm. An application: tailored Web site mapping. In *WWW*, pages 317–326, 1998.

[17] Hoong Chuin Lau, Trung Hieu Ngo, and Bao Nguyen Nguyen. Finding a length-constrained maximum-sum or maximum-density subtree and its application to logistics. *Discrete Optimization*, 2006.

[18] Hongyu Liu, Jeannette Janssen, and Evangelos Milios. Using HMM to learn user browsing patterns for focused web crawling. *Data & Knowledge Engineering*, 59(2):270–291, 2006.

[19] Gautam Pant and P. Srinivasan. Link contexts in classifier-guided topical crawlers. *IEEE TKDE*, 18(1):107–122, 2006.

[20] Gautam Pant and Padmini Srinivasan. Learning to crawl: Comparing classification schemes. *ACM TOIS*, 23(4):430–462, 2005.

[21] Jason Rennie and Andrew Kachites McCallum. Using Reinforcement Learning to Spider the Web Efficiently. In *ICML*, 1999.

[22] Michel Tokic. Adaptive ε-greedy exploration in reinforcement learning based on value differences. In *KI*, 2010.

[23] Jeffrey Travers and Stanley Milgram. An experimental study of the small world problem. *Sociometry*, 34(4), December 1969.

[24] Twitter. GET statuses/user_timeline. `https://dev.twitter.com/docs/api/1.1/get/statuses/user_timeline`, 2013.

[25] Jaewon Yang and Jure Leskovec. Patterns of temporal variation in online media. In *WSDM*, pages 177–186, 2011.