Exploration adaptative de graphes sous contrainte de budget

Georges Gouriten* Silviu Maniu†
Pierre Senellart*†
*Institut Mines–Télécom; Télécom ParisTech; CNRS LTCI, Paris, France
†The University of Hong Kong, Hong Kong

Nous nous intéressons dans cet article à l'exploration d'un graphe tel celui du Web ou d'un réseau social dans un contexte où les nœuds (et les arêtes qui en sont issues) sont découverts un à un, et où le nombre total de nœuds que l'on peut explorer est contraint. Le but est d'optimiser un score global du sous-graphe découvert, fonction monotone de scores élémentaires sur chaque nœud. Ce problème se pose en particulier quand on souhaite collecter les pages du Web correspondant à un sujet donné ou quand on utilise l'API du site d'un réseau social tel Twitter pour constituer un jeu de données centré sur d'un thème. Nous présentons une abstraction de ce problème faisant appel à deux composants principaux : une stratégie d'exploration et un estimateur du score des nœuds de la frontière du graphe. Nous montrons qu'une stratégie gloutonne est suffisante en pratique, et qu'il est possible de s'adapter aux caractéristiques de différents graphes en utilisant des estimateurs qui apprennent automatiquement les caractéristiques prédisant le mieux les scores des nœuds. Ces techniques sont appliquées à des des graphes réels issus de Wikipedia ou de Twitter.

1 Introduction

Focused crawling consists in crawling resources that are considered as relevant from a specific point of view. Those resources can be the blog posts related to the 2013 Bulgarian parliamentary election, for an archivist from the National Library of the Republic of Bulgaria, or the tweets talking about music in a social network.

To have good focused crawling systems is *important* and is *difficult*. It is important because the volume of the resources we can access is *booming*, whether it is Web pages, data from Web services, or data from any digital media (e.g., TV, radio). Data access is also generally becoming easier, thanks to bigger bandwidth and better hardware. However, access to resources is still not free and some of them are still very expensive, for instance with limitations on Web service APIs (e.g., Twitter only allows a fixed number of requests in a 15-minute period on its various APIs [26]).

It is also difficult as it consists in expanding a subset of resources with an estimation as good as possible of the quality of the new resources crawled. This estimation is often very difficult to make given the quality and quantity of information provided.

In this work, we propose a simple and convenient abstraction for focused crawling, and then propose observations and systems adaptable to various use cases.

We start with a wide formalization of the problem in Section 2 that, unlike the other works on focused crawling so far, can fit various settings, such as a web page crawl as well as a social network crawl through API accesses.

In Section 3, we introduce the main high-level algorithm for focused crawling and its separation of concerns. In the next section, Section 4, we discuss the crawler steering in a scenario where we would know the frontier. In Section 5, we study how to estimate the quality of the nodes at the frontier. We then investigate how we can combine a steering strategy and an estimator in order to achieve the most performing crawler. Just before concluding, we detail the related works.

2 Model and Problem Definition

In this section, we present an abstraction of the focused crawling problem in the general setting of node- and edge-weighted graphs. We start with some formal definitions, then move to the problem definition before introducing some use cases that go beyond Web crawling.

2.1 Formal Definitions

Our general setting is that of a directed graph $G = (V, E)$ where V is the set of vertices and $E \subseteq V^2$ the set of edges, with nonnegative weights on both edges and nodes. Nodes of the graph G stand for resources that we want to crawl (e.g., Web pages), edges for hyperlinks between resources (e.g., Web links), node weights for scores of relevance of resources (e.g., how much the Web page is in the scope of the current focused crawl), and edge weights for a priori indications of the relevance of the target node (e.g., how much the hyperlink occurs in a context that is in the scope of the current focused crawl).

We will need the notion of *frontier* of a graph, given a subset of its vertices:

Definition 1. Let $G = (V, E)$ be a graph and V' a subset of V. The *frontier of G induced by V'*, denoted $\text{Frontier}_G(V')$, is the subset of V defined by:

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

The notion of frontier is used to describe what constitutes a valid *crawl sequence*:

Definition 2. Let $G = (V, E)$ be a graph and V_0 a non-empty subset of V. A *crawl sequence starting from V_0 in G* is a non-empty sequence $(v_1 \dots v_n) \in V^n$ of vertices such that for all $1 \leqslant i \leqslant n$, $v_i \in \text{Frontier}_G(V_0 \cup \{v_1 \dots v_{i-1}\})$.

A crawl sequence describes how we progressively construct a crawled subgraph by adding nodes from the frontier, one at a time.

We now explain what we mean by edge weights:

Definition 3. Given a graph $G = (V, E)$, an *edge scoring function* on G is a function $\alpha : E \to \mathbb{Q}^+$. It assigns a score (or weight) to every edge in the graph.

For node weights, instead of a function that maps nodes to scores, we provide more generally a function that maps node sets to scores:

Definition 4. Given a graph $G = (V, E)$, a *subset scoring function* is a function $\beta : 2^V \to \mathbb{R}^+$. It assigns a score to a subset of V. For convenience, $\forall v \in V, \beta(\{v\})$ — abusively noted $\beta(v)$ — is called the *weight* of the node.

If one is given a node scoring function, it is easy to construct a relevant subset scoring function – for example, take the subset weights to be the sum of node weights of the subset, or the count of nodes with a positive weights. These two example exhibit a *monotonicity* property that is critical in the subset scoring functions we consider, allowing incremental computation of scores of subgraphs.

Definition 5. A subset scoring function β is *monotonically increasing* if and only if:

$$\forall (V', V'') \in (2^V)^2, V' \subseteq V'' \implies \beta(V') \leqslant \beta(V'').$$

In a graph with a (monotonically increasing) subset scoring function, our objective is simply to find an *optimal* crawling sequence of a given length, in the following sense:

Definition 6. Given $G = (V, E)$, β a monotonically increasing subset scoring function on G, $V_0 \subseteq V$, and n a positive integer called the *crawl budget*, we define the *optimal crawl sequences of length n starting from V_0* as:

$$\underset{\substack{(v_1 \dots v_n) \\ \text{crawl sequence from } V_0}}{\arg\max} \beta\left(V_0 \cup \{v_1 \dots v_n\}\right)$$

The unique β value of all the optimal crawl sequences is the *optimal score*.

Note that α, the edge scoring function, does not appear in this definition. Though edge weights take no part in our problem definition, they will be used as proxies for unknown node weights in the crawling algorithms we present further on.

The crawl budget is typically implied by *rate limitations* on the number of crawling steps we are able to perform, together with a total time allowed for the full crawl. To be more precise, assume that we are allowed k requests for each period of duration d (on some of Twitter APIs [26], for instance, $k = 300$ and d is a 15-minute period), and a total crawling time of T (e.g., one day or one week). Then n can be computed as $\frac{kT}{d}$.

In practice, we will see that finding an optimal crawling sequence is impractical for the following reasons:

1. In concrete scenarios, we do not have *full knowledge* of the whole graph, but we discover it as we crawl, *online*.
2. Even assuming full knowledge, determining an optimal sequence (or, even more simply, determining the optimal score) is intractable (see Section 4.3).
3. An algorithm for determining a crawling sequence has to respect crawling rate limitations: every period of duration d, it should provide k next nodes to crawl, in order to make maximal use of the rate limitation.

Consequently, we will aim at finding a crawl sequence with as high a score as possible, rather than an optimal one, and that conforms to the rate limitations.

2.2 General Use Cases

We now explain how, for a variety of problems, V, E, α, β, V_0, and n can be instantiated. The goal is to show that our abstract model covers different use cases, beyond classical focused Web crawling. To simplify, in this section we define β on individual nodes and assume that some monotone function (e.g., sum) is used to obtain a subset scoring function.

Focused Web crawling [7]. This is the classical focused crawling scenario: find, while crawling, Web pages matching as best as possible a keyword query.

V is the set of Web pages;
E is the set of hyperlinks;
α is the relevance of the anchor text of the hyperlinks to the focusing keywords;
β is the relevance of pages with respect to the focusing keywords (e.g., as mesured by tf–idf);
V_0 is an initial seed list of Web pages, e.g., manually designed;

n is the number of HTTP requests our bandwidth allows for (e.g., $k = 1{,}000$ requests per d of 1 second, spread over the thousand processing queues of the crawler; if T is one week, $n \approx 600{,}000{,}000$).

Topic-centered Twitter user crawl [16]. Here, the goal is to find the Twitter users that are the most relevant with respect to a keyword query, i.e., whose tweets are collectively the most relevant.

V is the set of Twitter users;
E is the tweet *mentioning* relation: $(u, v) \in E$ if a tweet of user u mentions one of user v;
α is the relevance of tweets in the communication history of the users;
β is the relevance of a user's tweets with respect to the focusing keywords;
V_0 is an inital seed list of users, e.g., obtained using the Twitter Search API for this given query;
n is the number of `statuses/user_timeline` requests Twitter's API allows us to perform – accounting for a rate limitation of $k = 300$ requests per d of 15 minutes [26], and T being one week of crawl, it means for instance $n \approx 200{,}000$.

Deep Web siphoning [3] through a keyword search interface. The goal is to siphon an entire deep Web database, accessible beyond an HTML form, using keyword queries through the form, discovering new keywords in response pages in the query.

V is the set of keywords;
E is such that $(u, v) \in E$ if the keyword v appears in the response page obtained by submitting the form with keyword u;
α is the number of occurrences of the keyword v in result pages for keyword u;
β is the number of records returned for keyword a;
V_0 is an initial small dictionary of keywords;
n is the number of HTTP requests crawler politeness constraints allow (say, 600,000 for a week of crawling if one second between requests in an acceptable delay).

Beyond these classical crawling examples where the crawling entity is centralized, there are also cases where the crawling process is distributed:

Gossiping peer-to-peer search [2] in, say, a file sharing application. One peer issues a query and this query is propagated through gossiping to neighboring peers.

V is the set of peers;
E is the peer-to-peer overlay network;
α is the relevance of the cached information about a remote peer to the query;
β is the relevance of a peer's data to the query;
V_0 is the peer issuing the query;
n is the total number of propagation steps allowed as prevention against flooding over the network (e.g., one tenth of the total number of nodes of the network, say $n = 10{,}000$).

Using a real-world social network to answer a query [25, 13]. This is a classical sociological experiment where an individual is asked to use its direct social network to, e.g., forward a message to another person of the network that he is not directly connected to.

V is the set of individuals;
E is the friendship/acquaintance network;
α is the assessment of an individual of her acquaintance's expertise on a query;
β is the self-assessment of an individual of her expertise on a query;
V_0 is the user the query starts from;

n is the collective effort, in terms of requests made from an individual to another, that we allow for a given query (say, we stop the experiment when 1,000 individuals have contributed).

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

We claim that our general framework, and the algorithms we present in the next few sections, can accordingly be used in a wide range of scenarios and are a basis for building efficient systems that deal with one of the cases above. Obviously, specific settings may also require specific adjustments; in particular, we won't mention further the problem of managing distributed crawls.

2.3 Experimental Use Cases

We focus in this paper on the first two use cases above (focused Web crawling and topic-centered Twitter user crawl), experimenting over five different datasets with different characteristics in the goal of testing the robustness of different approaches to budget-constrained graph exploration. All datasets are available upon request.

Instead of raw occurrence scores for nodes and edges, we use a logarithm smoothening, defined by the function: $f : x \mapsto \begin{cases} 1 + \log(x) & \text{if } x > 0 \\ 0 & \text{otherwise} \end{cases}$; the use of f is detailed further. We also experimented with variations of the scoring mechanism with little impact on the results.

Wikipedia datasets To simulate a Web crawling scenario, we build two different datasets from the content of the French Wikipedia[1]. The nodes are Wikipedia pages, the edges links across pages (hence, the obtained graph is an induced subgraph of the Web graph). Node and edge scores depend on a keyword query (we use *bretagne* in one dataset, to obtain a relatively small and focused set of nodes with positive score, *france* in the other). More precisely, if x is the number of occurrences of the keyword in a page u, $\beta(x) = f(x)$; if y is the number of occurrences of the keyword in a 100-character window around a hyperlink from u to v, $\alpha(u, v) = f(y)$.

Twitter datasets We build three graph exploration datasets out of the SNAP Twitter data [27][2], which contains an estimated 20–30% of all tweets published between June and December 2009, around 476 million tweets for 17 million users, with 93 million *mentioning* links across tweets (mentions include tweets that reply to a user, retweets, and other cases where a user cite another one in her tweet). Our Twitter datasets are also based on a keyword query (respectively, *happy*, *jazz*, *weird*). Nodes correspond to users, edges to mentions between tweets of this user. Again, if x is the number of occurrence of a keyword in the tweets of a given user u, $\beta(x) = f(x)$; if y is the number of occurrences of the keyword in the tweets of user u mentioning user v, $\alpha(y) = f(y)$.

For both datasets, we experiment with various ways of combining node scoring function into node subset scoring functions, but mostly use sum, unless otherwise specified.

Statistics of all five datasets are detailed in Table 1. In particular we mention the number of nodes and edges with non-zero score. Observe the diversity of sparsity in the different datasets: *jazz* or *bretagne* have a very small portion of non-zero nodes among those of the original graph, while *france* and *happy* cover a much more significant amount of nodes, with *weird* somewhere in between.

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

[2] This dataset was formerly available at `http://snap.stanford.edu/data/twitter7.html`. As per Twitter's request, it is no longer publicly available.

Table 1: Dataset statistics

Origin	Keyword	Non-zero nodes	Non-zero edges
Wikipedia	bretagne	42,923	163,017
	france	2,162,969	2,410,783
Twitter	happy	1,860,131	1,898,015
	jazz	105,240	50,629
	weird	541,937	342,153

3 High-Level Algorithm

We present a high-level view of a generic algorithm for determining a crawl sequence that aims to be as close as possible to the optimal one; this algorithm depends on two main components: an *estimator* that computes an *estimated score* for all nodes of the frontier and a *strategy* that, given this estimated score, picks the next node(s) to crawl. Let us first see these two components as black boxes. Potential strategies and estimators will be discussed and compared in, respectively, Sections 4 and 5.

Algorithm 1 describes this algorithm, it takes as input an initial seed graph G_0 (i.e., the subgraph of the full graph induced by a set of seed nodes V_0), a budget n, and a refresh rate r. It returns a crawl sequence V whose score is chosen as close as possible to the optimal one.

The need for a *refresh rate* comes from situations where estimations can take a long time compared to other steps. We might then want to refresh our estimation not at each step but every r steps, r is the *refresh rate*.

Algorithm 1: Main crawling algorithm whose behavior depends on two black boxes: *strategy* and *estimator*

input : a seed graph G_0, a budget n, a batch size k
output : a crawl sequence V

```
1  V ← ();
2  G' ← G_0;
3  for i ← 1 to n/r do
4  |   frontier ← extractFrontier(G');
5  |   scoredFrontier ← estimator.scoreFrontier(G', frontier);
6  |   NodeSequence ← strategy.getNextNodes(scoredFrontier, r);
7  |   V ← (V, NodeSequence);
8  |   for u in NodeSequence do
9  |   |   G' ← G' ∪ crawlNode(u);
10 return V
```

The algorithm maintains a crawled subgraph G', initialized to G_0 (line 2) and updated as more nodes are crawled (lines 8–9). The main loop of the crawl (lines 3–9) iterates over $\frac{n}{k}$ batches of size k. First, the frontier is extracted from the crawled graph (line 4) – in reality, the frontier is maintained from one iteration to another to improve efficiency. Our first black box, the estimator, is then used to score the frontier: assign a predicted score to all nodes of the frontier. For that purpose, the estimator has access to the whole crawled graph that can serve as a training set. Some of the estimators we present in Section 5 indeed make use of such a training. The next line calls the strategy black box to get the next k nodes to crawl according to the scored frontier – as we explain in Section 4 we can in practice be *greedy* and take the k nodes with the highest score. The node

sequence is updated accordingly (line 7).

Again, the pseudo-code is a simplified view of our algorithm that does not include any of the optimizations we use to avoid, e.g., retraining the estimator at each step on the whole crawled graph. We briefly discuss our implementation in Section 6.1.

With this generic crawling algorithm in mind, we now only need to specify which crawling strategy to use (Section 4) and which estimators yield best results (Section 5). In Section 4 we also investigate the impact of k on the performance of the algorithm. We finally put everything together and test the full system in Section 6.

4 Strategies: Steering the Crawler

This section gives us some intuition on the strategy black box, i.e., on how to steer the crawler in the graph. To perfectly decouple the analysis of the strategy and of the estimator used, we suppose in this section we have access to an *oracle estimator* that is be able to perfectly estimate nodes of the frontier. In the following section, we will see how we can build estimator to approximate this oracle.

We first investigate the performance of the greedy strategy, then investigate the impact of the refresh rate, and we conclude by investigating whether an optimal strategy is possible.

4.1 Rich Friends Will Make You Richer

Should we always pick the best node? Interesting parts of the graph could be missed because of too systematically greedy a strategy. A node bridging to a rich subgraph could have a low score and thus be missed. In order to have some intuition on the importance of this risk, we compared the greedy strategy, i.e., always selecting the next best node, with *altered greedy* strategies that introduce some randomness.

Definition 7. The *greedy* strategy consist in crawling at each step

$$\underset{v \in \text{Frontier}_G(V')}{\arg\max} \beta(v).$$

Definition 8. The *altered greedy* strategy consist in setting two parameters, $q \in [0, 1]$ and $\zeta \in \mathbb{Q}^+$, then crawling at each step, with a probability q,

$$\underset{v \in \text{Frontier}_G(V')}{\arg\max} \beta(v)$$

and with a probability $1 - q$, a node uniformly at random among those in

$$\left\{ v \in \text{Frontier}_G\left(V'\right) \mid \max_u(\beta(u)) - \beta(v) \leqslant \zeta \max_u(\beta(u)) \right\}.$$

Experiments We evaluated a crawl sequence of 5,000 steps on the datasets presented in Section 2.3. We compared the greedy strategy and different altered greedy defined by their (q, ζ) . The results were averaged over 10 seed subgraphs of size 1,000, whose nodes were picked randomly among nodes with a score over 1.5.

The results for the `bretagne` and `jazz` datasets are presented in Figure 1. The x-axis shows the crawling budget, the y-axis the relative score achieved with respect to greedy. The scenario visible on these two graphs is the same for every graph, introducing some randomness does not hurt much until about 1,000 steps (it sometimes even beat greedy in other scenarios); however, in the long run, greedy is the winning strategy.

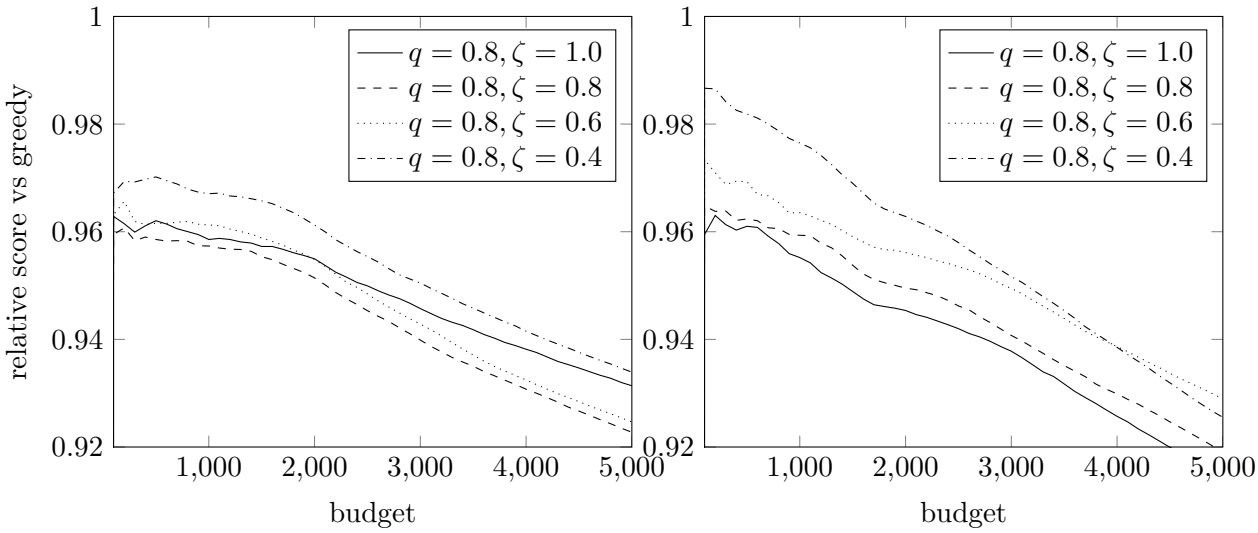


Figure 1: Greedy versus altered greedy for `bretagne` and `jazz`

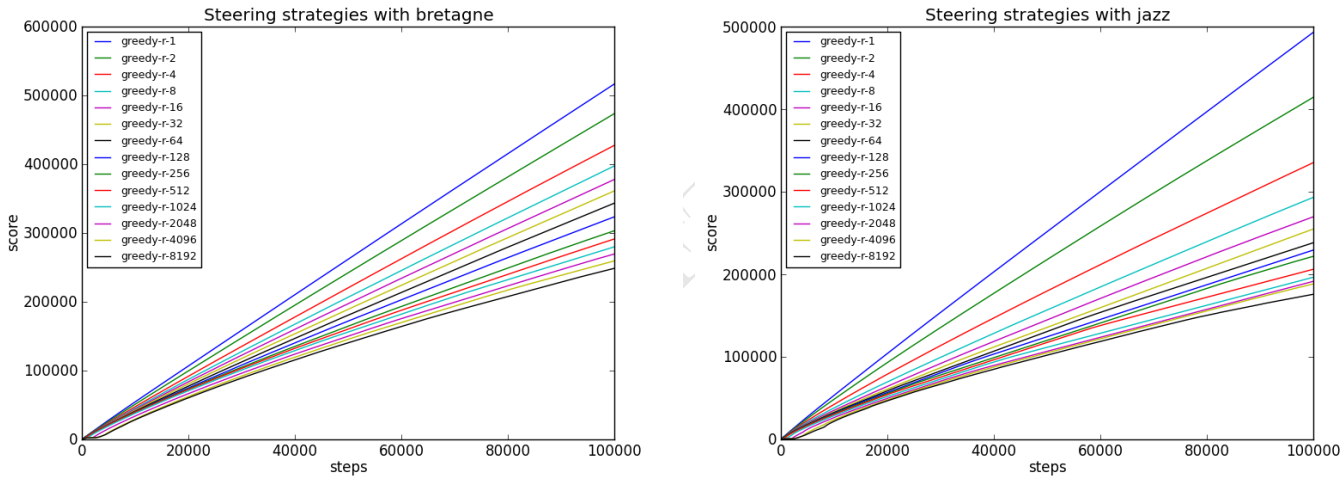


Figure 2: Influence of refresh rate on greedy for `bretagne` and `jazz`

This can be explained by the fact that the size of the frontier is relatively small at the beginning, thus some rich part of the graph can hide behind small score nodes. However, past 1,000 steps, the randomness introduced does not bring as much and slows down the crawl. We can see the good performances of greedy as a "rich friends will make you richer" property. As this holds in all five datasets, we will consider the greedy strategy as a given in the rest of our study.

4.2 The refresh rate advantage

We ran some experiments to see the impact of the refresh rate defined in Section 2 on the crawl performance. The setting is the same as for the previous experience, with 100,000 steps instead of 5,000.

As we can see in Figure 2, a higher refresh rate has a significant long-term negative effect on the crawling performance. A refresh rate of 2 degrades the performances by $\approx 5\%$, one of 500 by $\approx 20 - -50\%$. This is the *refresh rate advantage*.

4.3 A baseline hard to find

One would like to compare the crawl sequence resulting of an online focused crawl with the optimal crawl sequences as defined in Section 2. Unfortunately, even in an offline setting with full knowledge (i.e., where it is possible to access the whole graph), even determining the optimal cost is NP-hard, even if the subset scoring function is the sum of individual node scores.

Proposition 1. *Given a graph G, a PTIME-computable subset scoring function β, a subset of vertices V_0, and a budget n, determining if there is a crawl sequence of score greater than or equal to a given rational r is an NP-complete problem. NP-hardness holds even if β is obtained by summing up individual node scores, if V_0 is a singleton, and if r is an integer.*

Proof. Membership in NP is straightforward: guess a node sequence of size n, checks 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 the NP-hardness, we exhibit a PTIME many-one reduction from the LST-Graph problem described in [18]. The edge-uniform LST-Graph problem is defined as follows: given two 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 [18] shows this problem is NP-hard, by reduction from Set-Cover.

Let (L, W, V, E, w) be an instance of LST-Graph. Without loss of generality, we can assume L to be at most $|E|$ (otherwise, just set L to $|E|$, the problem will have the same answer). 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 $u_1, \ldots, u_{L+1}$;
- for each node u of G, a chain of $L + 2$ edges $(r, u_1), (u_1, u_2), \ldots (u_{L+1}, u)$.

Since L is at most $|E|$, this construction is in $O(|V| \times |E|)$. We set $\beta(X) = \sum_{u \in V \cap X} w(u)$ (in other words, the score of a node is the sum of scores of all nodes of the old graph, new nodes having score 0).

We claim that edge-uniform LST-Graph(L, W, G, w) has a solution if and only if $(G', \beta, \{r\}, 2L+1)$ 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 edge-uniform LST-Graph. Let T be a subtree of G of total weight at least W and of length l at most L. Let $(v^1 \ldots v^m)$ 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 $(v^1_1, \ldots, v^1_{L+1}, v^1, \ldots, v^m)$ 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 $u_1, u_2, \ldots, u_{L-l}$ for an arbitrary node $u \in V$ distinct of v^1. The score of S is exactly the summed weight of T, and is thus $\leqslant W$.

Conversely, assume $(G', \beta, \{r\}, 2L+1)$ 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 the first node u in r, S such that $(u, v) \in G'$ to v. Consider the forest $F = T \cap V$. F is a forest of G of length at most L (because if F is not empty, T must include at least one chain $v_1, \ldots, v_{L+1}, v$) 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 $u_1, \ldots, u_{L+1}, u$ and $v_1, \ldots, v_{L+1}, v$ 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. □

This result means there is no practical way to determine whether a heuristic strategy is close enough to the optimal. In particular, the greedy heuristic cannot be optimal in the general case. As

in [18], we leave as an open problem the possibility of approximating the optimal score or of finding an crawling sequence with score a factor of that of the optimal one.

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

5 Estimating Frontier Nodes

For any crawling campaign, its objective is to retrieve the most relevant nodes (by their β value). As shown in Section 4, a greedy approach can be used, in which at each step one chooses the node having the highest *estimation* of its β value. It becomes thus crucial to be able to have good estimators of node weights. We present below the estimators covered by our study.

In this section, we will use the following notation. We define V' as a crawled set of nodes, E' the known edges – including $\text{Frontier}_G(V')$ –, $d_o : V \to \mathbb{N}$ (or d_i) the number of known outgoing (or, respectively, incoming) edges $e \in E'$ of a node, and $\widetilde{\beta} : \text{Frontier}_G(V') \to \mathbb{Q}^+$ the weight estimation function.

5.1 Baseline Estimators

The simplest baseline estimator simply chooses a random node in the frontier:

Estimator 1. RANDOM The *random (R)* estimator is defined by $\widetilde{\beta}(v) = random(\mathbb{Q}^+)$.

In practice, this gives very poor performance. A slightly better baseline estimator is based on the classical crawler behavior, visiting the nodes in breath-first manner:

Estimator 2. BREADTH-FIRST The *breadth-first (BF)* estimator is defined by $\widetilde{\beta}(v) = \frac{1}{l(v)+1}$, where $l(v)$ is the distance of a node $v \in V'$ to V_0.

5.2 Neighborhood-based Estimators

The second type of estimators we study are estimators based only on the *neighborhood* of the nodes. More precisely, we study estimators which make estimations based on a linear combination of *features* of frontier nodes.

The features are based on the immediate neighborhood of the nodes of the frontier, more precisely the *incoming* edges and nodes. The incoming information represents the only information we have of the frontier nodes – their weight and outgoing edges are retrieved only when crawled. Although one can go "deeper" in the subgraphs, by creating features which takes into account weights of siblings, ancestors of incoming nodes, etc., we restrict ourselves to first-level features in this study. The reason is that, during our experimental evaluation, we found that increasing the number of features in neighborhood-based estimators actually decreases the quality of the estimation.

More precisely, the three features based on incoming links are:

- nodes: $f_n(v) = h(I_n)$, where $I_n = \{\, w(u) \mid (u, v) \in E \,\}$,
- edges: $f_e(v) = h(I_e)$, where $I_e = \{\, w(e) \mid e = (u, v) \in E \,\}$,
- nodes-edges: $f_{ne}(v) = h(I_{ne})$, where $I_{ne} = \{\, w(u) \times w(e) \mid e = (u, v) \in E \,\}$,

where h can be any monotone function, such as `sum`, `avg`, or `max`.

Table 2: Pearson correlation coefficients between node scores and features.

Dataset	Type	n_sum	e_sum	ne_sum	n_max	e_max	ne_max
france	in	0.118	0.252	0.248	0.357	0.272	0.269
	out	0.639	0.827	0.687	0.313	0.344	0.295
bretagne	in	0.315	0.238	0.404	0.195	0.309	0.300
	out	0.570	0.828	0.641	0.262	0.326	0.322

Table 3: Fit coefficients (R^2) for linear models.

Dataset	Type	sum	max
france	in	0.121	0.145
	out	0.695	0.152
bretagne	in	0.191	0.129
	out	0.688	0.151

Estimator 3. LINEAR FIRST-ORDER The *linear first-order neighborhood (LNH)* estimator is defined by $\widetilde{\beta}(v) = c_n \times f_n(v) + c_e \times f_e(v) + c_{ne} \times f_{ne}(v) + c_0$, where c_n, c_e, c_{ne}, c_0 are real-valued coefficients.

In its simplest form, the first-order estimator will only use one feature , i.e., one and only one of c_n, c_e, c_{ne}, is non-zero. In this case, the estimation simply becomes a function of a combination of incoming nodes and edges. For example, when $c_n = 1$ and $c_e = c_{ne} = c_0 = 0$ and $h = \texttt{sum}$, the estimator is based solely on the sum of incoming node weights. As our experiments show, this type of estimator can perform surprisingly well, depending on the type of network. There is, however, no clear cut best estimator for all the datasets we have studied, making them hard to use in practice, when information about the network is missing.

An adaptive estimator can be easily designed by considering the already crawled nodes as training data. At each step (or several steps) a *least-squares linear regression* is performed on these training data, thus determining the c_n, c_e, c_{ne} and c_0 coefficients. This estimator is quite efficient in practice, even when using open-source solvers, and can be performed incrementally, without the need of full retraining when new training data is available.

Correlation

5.3 Subgraph-based estimators

As an alternative to first-order estimators, which only use the information about the nearby nodes, some related work consider subgraph-based estimators.

Navigational Rank Navigational Rank [15] is a two-step page importance computations specifically designed for focused crawling. The first step is an iterative propagation from the offspring to the ancestors, combined with the actual node score:

$$NR_1(v)^{t+1} = d \times w(v) + (1-d) \times avg_{(u,v)\in E} \frac{NR_1(u)^t}{indeg(u)}$$

where $NR_1(u)^t$ is the node score at iteration step t, and d is a parameter

The second propagation step is performed only on frontier nodes, and takes place from ancestor to offspring, as follows:

$$NR_2(v)^{t+1} = d \times NR_1(v) + (1-d) \times avg_{(v,u)\in E} \frac{NR_2(u)^t}{outdeg(u)}.$$

If we consider the NR_2 scores as the estimation of node values, we obtain the following estimator:

Estimator 4. NAVIGATIONAL RANK The *Navigational Rank (NR)* estimator is defined by $\widetilde{\beta}(v) = NR_2(v)$.

We have tried testing with respect to this particular estimator; however, Navigational Rank proved much too slow to be compared with respect to any of the other estimators – simply simulating a few thousand steps of crawls took hours, with respect to seconds for all other ones. Indeed, one step of Navigational Rank estimation requires two successive iterative computations (with perhaps a few dozens steps) on the whole crawled graph. Since this needs to be repeated at every crawling step, this results in an overall quadratic complexity for Navigational Rank estimation during a full crawl. Consequently, we were not able to provide any experimental comparison between Navigational Rank and other estimators. Note that the analysis performed in [15] considers graphs – typically graphs of Web sites – much smaller than those we consider in this work.

OPIC The OPIC [1] system is an online page crawling system designed to maximize the retrieval of high PageRank pages on the Web.

For this, during the crawl, OPIC maintains two per-node counters: $C(v)$ - the *cash* value of of a node, and $H(v)$ its cash history. It also maintains a global counter G for the entire cash accumulated in the system.

The estimation is made at crawl time, in a three step approach:

1. a node v is chosen for updating, and the 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 $C(u) = C(u) + \frac{C(v)}{outfdeg(v)}$,
3. the cash value of v is reset, and the global counter incremented $G = G + C(v)$, $C(v) = 0$.

Then, the estimation of a node's value is simply the estimation of its PageRank value, based on the three counters above:

Estimator 5. OPIC The *OPIC* estimator is defined by $\widetilde{\beta}(v) = \frac{H(v)+C(v)}{G+1}$.

The main disadvantage of the original computation of the PageRank measure in the OPIC system is that it takes in account only outlinks of pages, without taking into account the node and edge scores. To alleviate this disadvantage, in similar vein to the neighborhood based estimator, we experiment with the following propagation mechanism: along the outlinks the cash is distributed in a normalized proportion computed based on (i) the sum of edge weight, (ii) the outgoing node weight and (iii) the multiplication of node and edge weights.

5.4 Comparing Estimators on the Same Crawl

To compare estimators, we measured the difference of actual score of the top node ranked by a given estimator with respect to the score of top node ranked by an oracle evaluator, i.e., an estimator guessing the actual node score at each step. We have tracked the differences at each step between each neighborhood estimator and the oracle estimators, *when the crawl follows an oracle estimator.*

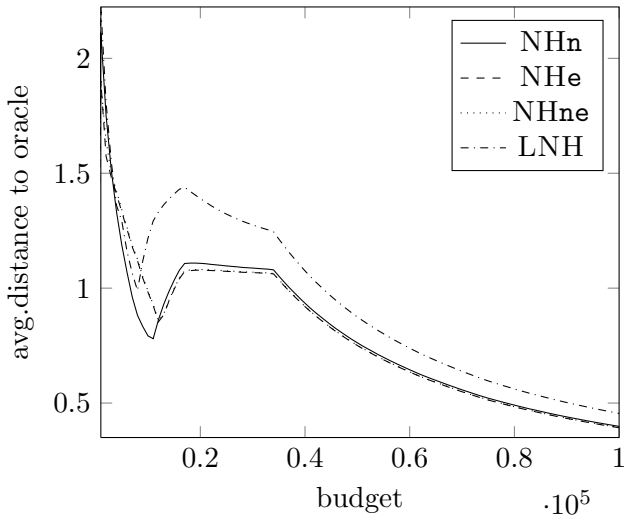


Figure 3: Estimator precision for `bretagne`.

We present the results for the Wikipedia *bretagne* graph (the other graphs are similar) in Figure 3. In order to smooth out the results, the rolling average with a window of 1,000 is plotted.

It can be seen that the most noise is encountered at the start of the crawl, and that all evaluators converge to choosing the best nodes towards the end of the crawl. Moreover, 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. Note however this is in a sense ideal behavior, when all previous steps have been chosen perfectly according to the oracle. This does show that the estimators – even simple ones – can be almost perfect estimators when the conditions are ideal. In the next section, we present more in-depth experiments on full, real-world crawls, which show that the behavior can vary significantly from this ideal case.

6 Full System Experiments

We now experiment with our complete system, incorporating main crawling algorithm, greedy strategy, and a variety of estimators as described in the previous section. First, we discuss our implementation, then our experimental setup, finally our results.

6.1 Implementation

Our system is implemented in C++, using in-memory storage of the graph (thanks to Boost graph library[3]) and crawler metadata. Each black box component is 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`). For linear regression, we use the dlib C++ library[4] that provides an *incremental* implementation of the linear version of the recursive least squares algorithm. Thanks to this incremental feature, we can maintain the model incrementally without having to retrain on the whole crawled graph at each step.

[3] `http://www.boost.org/libs/graph`
[4] `http://dlib.net/`

6.2 Experimental Setup

All experiments are run on a Linux PC with 48 GB RAM. In-memory storage of our datasets (especially the Twitter graph) does require a large amount of available memory.

For each dataset, we have randomly generated 100 seed graphs containing 50 seed nodes each, chosen from nodes having non-zero scores. For each such seed list, the initial graph is the subgraph induced by the 50 nodes. The initial frontier is the list of outgoing nodes of the 50 nodes. Each crawl is run for a budget of 100,000 steps. Considering that the current (at the time of writing) Twitter API request limit is 300 per 15 minute window [26], the budget would allow a focused crawl campaign of 3 days and a half, which is reasonable both in terms of time and freshness of crawled data. In the experiments, we assume that the computation score is dominated by the time between requests. For the crawl score, we use the `sum` and `count` aggregation functions of node scores.

6.3 Results

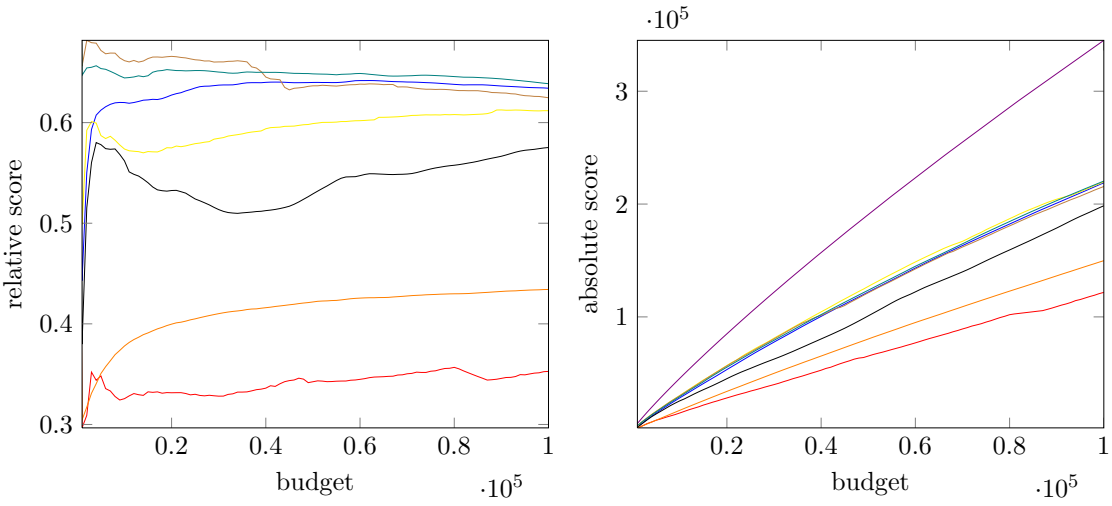
We present the results in Figure 4 for the Twitter graphs, and Figure 5 for Wikipedia. The left column shows the evolution of the score relative to a oracle estimator, which always chooses the node with the highest weight on the frontier, while the right column shows the evolution of the absolute score. To preserve legibility, we have chosen to only display a selection of the estimators, as they were either very similar to other estimators, or were not interesting to our discussion. Moreover, we have omitted the experimental results for `count`, since the results – and corresponding discussion – are equivalent to the `sum` case.

One can see that, quite surprisingly, the basic neighborhood estimators (NH) behave well in certain graphs. The estimators using nodes weights (NH`n`) or the multiplication of node and edge weights (NH`ne`), behave quite well, but which is the better one is not clearly established. Usually, the `ne` variant is better than `n`, except for the Wikipedia *bretagne* dataset. In our experiments, the variant based on edge weights, NH`e`, (not plotted) was virtually identical to the `ne` variant.

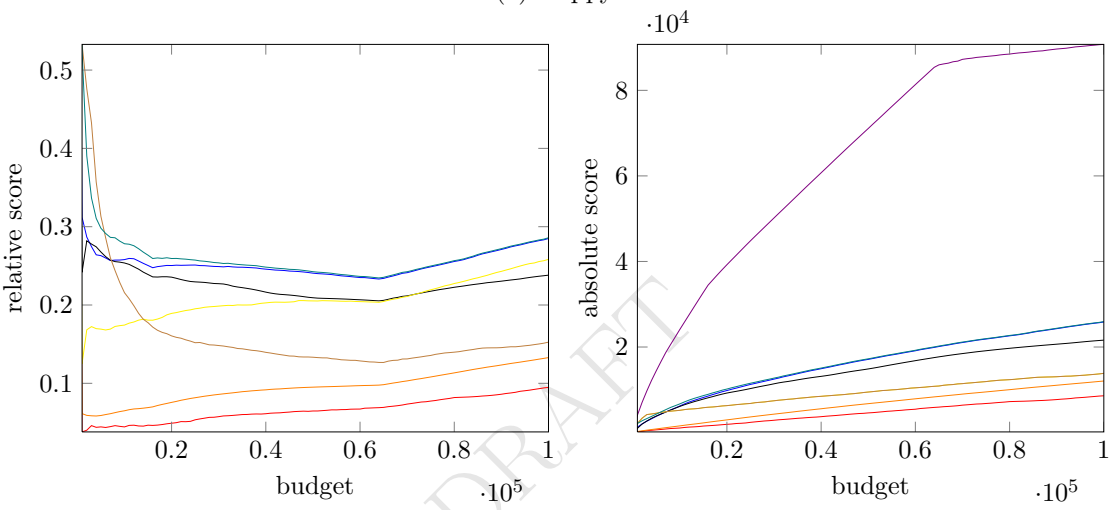
The linear regression based estimators, LNH, are plotted using two aggregations of their features: `sum` (e.g., the sum of incoming node weight) and `max` (e.g., the maximum value of the incoming node weights). They exhibit the same behavior, in that they are most times comparable with the simple neighborhood estimators, with a few exceptions: the Wikipedia *france* dataset for `sum`, and Twitter *happy* and *weird* for `max`. In general, `sum` is better on Twitter in the long term, in the late stages of the crawl, while `max` is better on the Wikipedia datasets.

Combining the two types of estimators leads to better results in general. More precisely, we start with the basic neighborhood estimators for the first step (1,000 in our experiments) and we switch to the linear neighborhood estimators after. The objective of this type of hybrid estimator is to allow the linear regression models to get enough training data before using them, as we observed that starting with few training data would lead to sub-optimal crawling sequences, even in the long term. Indeed, this type of combination leads to estimators which are always comparable to the best on *any* dataset, and sometimes even better than them – see Twitter *happy* for instance. Moreover, we noticed that it usually does not matter which starting estimator is chosen – out of the basic ones – as long as it is reasonable. Even if some of the neighborhood estimators might exhibit better potential on some datasets and for low budgets, the clear advantage of this estimator is that it is adaptive to any dataset, and can be safely used for such a constrained exploration.

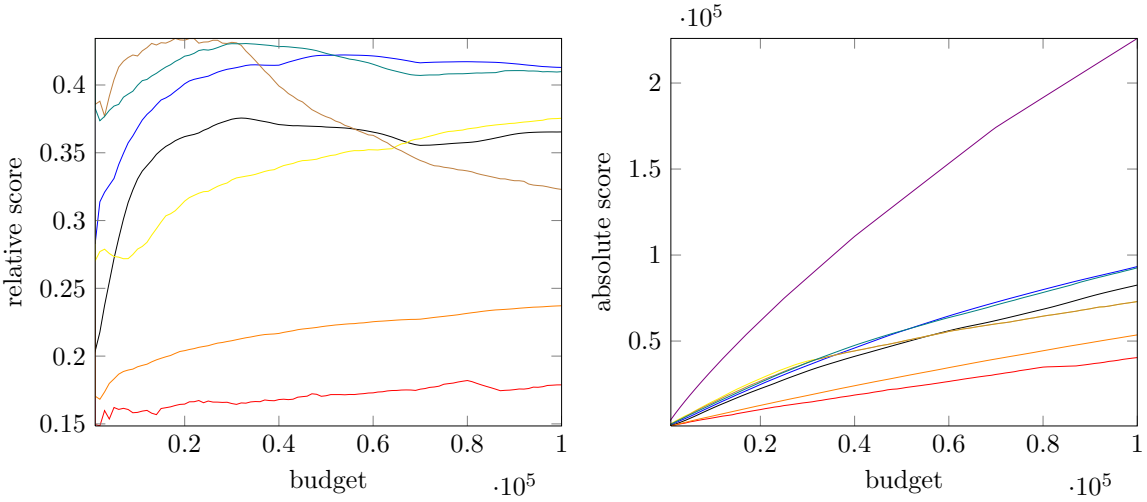
Finally, breath first exploration and the sub-graph based estimators – such as the plotted OPIC – are clearly worse overall than the neighborhood estimators, but represent interesting baselines. In the case of OPIC, the versions which propagate on edges and/or nodes were not better than the original version. Note that we could not reliably plot the Navigational Rank estimator, as its computation costs – even for low budgets – were extremely high, compared with all the other estimators.



(a) Happy



(b) Jazz



(c) Weird

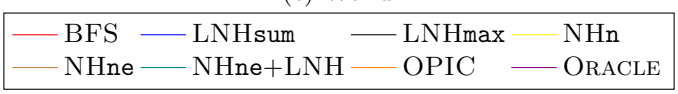
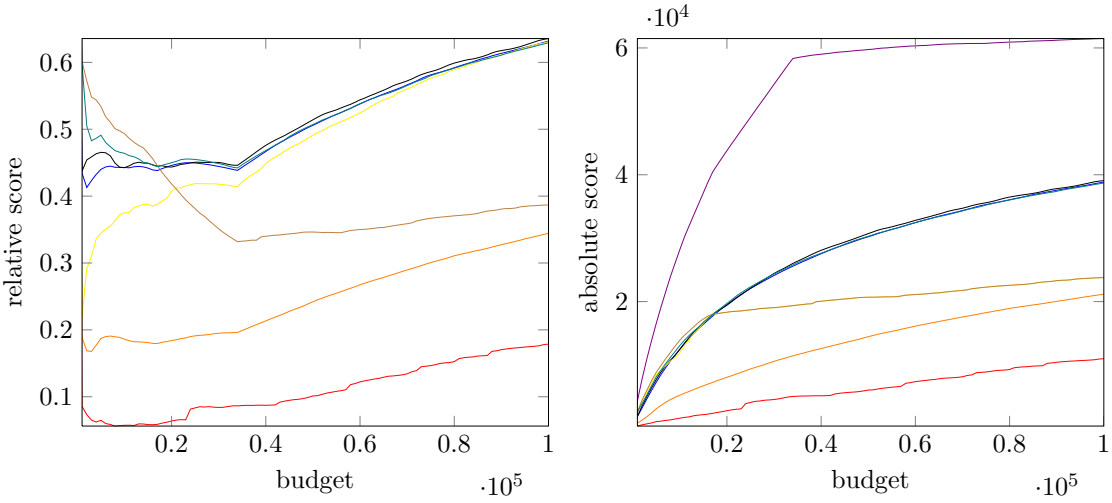
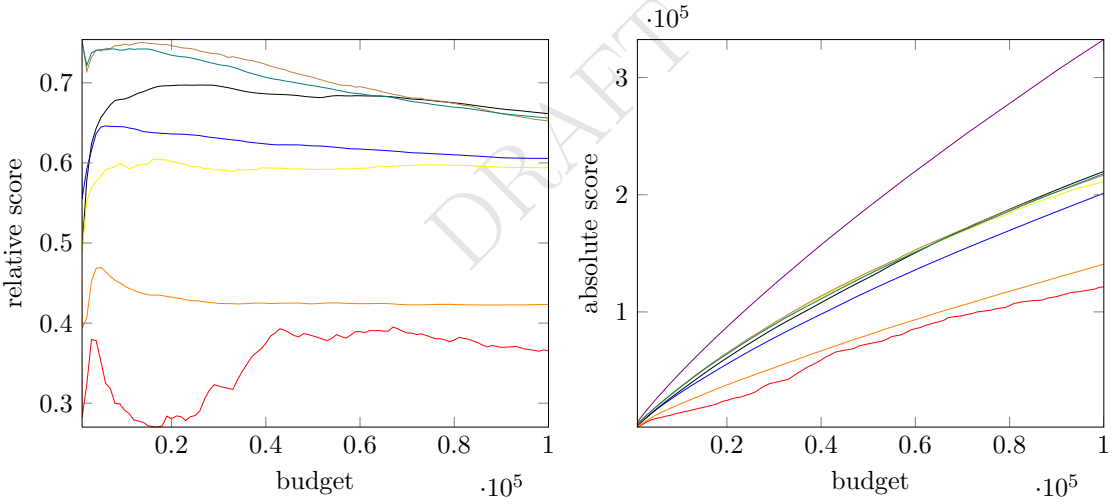


Figure 4: Results on the Twitter datasets



(a) Bretagne



(b) France

Figure 5: Results on the Wikipedia datasets

7 Related Work

7.1 Online problem

The first formulation of the concept of *focused crawling* is in the context of Web pages crawling and is given by Menczer [20]. It is then extended in [8, 6], where is described a crawler made of a classifier, to distinguish relevant pages, and a distiller, to identify pages more likely to point to other relevant pages.

All works on focused crawling so far have been Web pages centric. They consisted in improving Web pages and links scoring techniques on one hand, and looking at more generic PageRank-like ideas on the other.

We looked at the problem from a higher level perspective and used a formalization that allows us to consider more use cases than Web pages crawling.

Web pages and links scoring Among the first works in this direction are the "first-best" crawlers, represented by the Fish-search [11] and Shark-search [17] algorithms for scoring pages and links, based on the *tf-idf* measure of the textual content of pages. To improve their performance, more involved classifiers have been proposed based on Naive Bayes [8], neural networks and SVM [21], Hidden Markov Models [19, 4], and reinforcement learning [23]. The idea of using *context graphs* to better score pages and links was also formulated [12].

An experimental study on link context approaches has been performed in [21], and found that a crawler utilizing both the link context and page score significantly outperforms only link context approaches. For more details on focused crawling classifiers, we refer the reader to the survey in [22].

In our examples, we use different edge and node scoring functions with different outcomes. It allows us to study the generic graph problems and develop techniques that can be used in various cases.

PageRank-like approaches PageRank and related measures are proven to effectively represent the relative importance of pages on the Web. As a consequence, focused crawling can be also seen as an effort to retrieve such pages early and with limited resources. The first approach to estimate the PageRank value for such graphs is the OPIC system [1]. The RankMass crawler presented in [9] has as objective to maximize the coverage of important pages – defined in terms of *rank mass*. The Navigational Rank [15] measure is defined as a two-way page importance propagation, and is shown to outperform approaches such as context graphs.

We used those works has a baseline and extended them to take into account preexisting weights on nodes and links.

7.2 Offline problem

In the offline setting, our problem can be reduced to a graph exploration problem where we want to optimize the aggregated node weight on a size-constrained forest that covers the seed nodes. By reduction to the problem in [18], we show in Section 4.3 that it is NP-hard to find an optimal solution.

Note that, in the unconstrained case, when the budget is at least as high as the number of nodes in a graph, and when the full graph is known, the problem can be reduced to finding an optimum branching in the graph [10, 14], which admits efficient polynomial solutions [5, 24]. Informally, the optimum branching can be thought of as the extension of spanning trees to directed graphs.

8 Conclusions

We started our work modeling our problem as a graph exploration problem. The nodes are the resources, weighted with their content relevance. The edges are the links between resources, weighted with their relevance as links. We saw how this can be applied to different use cases. We then introduced the main algorithms and explained how we can separate concerns into steering and estimation. We first demonstrated some interesting properties regarding steering, the "rich friends will make you richer" property, the refresh rate advantage, and the NP-hardness of the offline problem. Then, we discussed estimation options. We introduced new estimators and adapted state-of-the-art estimators to our formalization. Combining estimation and steering, we created performing and adaptable focused crawlers. Eventually, after discussing related works, we claim the importance of our contributions. Our formalization of focused crawling is new and can be adapted to various cases; we demonstrated useful properties regarding steering; we covered a wide range of options for estimators and we managed to build robust focused crawlers.

This work also opens interesting opportunities for further work on focused crawling. How to find an approximation of the optimal solution to the offline problem? What kind of dynamics for very short crawl? Can we imagine better option for combining estimators, for instance using multi-armed bandits? How can we integrate the idea of re-crawling some nodes at different times?

Acknowledgments

We are grateful to Bogdan Cautis for discussions about this research topic. This work has been partly supported by the European Union's Seventh Framework Programme (FP7/2007–2013) under grant agreement 270739. Finally, we thank the Futur et Rupture program of Institut Mines–Télécom for funding Georges Gouriten's PhD.

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] F. Bock. An Algorithm to Construct a Minimum Directed Spanning Tree in a Directed Network. *Developments in Operations Research*, 1971.

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

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

[8] 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.

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

[10] Y J Chu and T H Liu. On the shortest arborescence of a directed graph. *Science Sinica*, 14, 1965.

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

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

[13] 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.

[14] Jack Edmonds. Optimum branchings. *J. Res. Bur. Stand*, 71B(4), 1967.

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

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

[17] 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.

[18] 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.

[19] 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.

[20] Filippo Menczer. ARACHNID: Adaptive Retrieval Agents Choosing Heuristic Neighborhoods for Information Discovery. In *ICML*, 1997.

[21] Gautam Pant and P. Srinivasan. Link contexts in classifier-guided topical crawlers. *Knowledge and Data Engineering*, 18(1):107–122, 2006.

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

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

[24] R E Tarjan. Finding optimum branchings. *Networks*, 7(1):25–35, 1977.

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

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

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