LEARNING TO SCHEDULE WEB PAGE UPDATES USING GENETIC PROGRAMMING

AÉCIO SOLANO RODRIGUES SANTOS

LEARNING TO SCHEDULE WEB PAGE UPDATES USING GENETIC PROGRAMMING

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação do Instituto de Ciências Exatas da Universidade Federal de Minas Gerais como requisito parcial para a obtenção do grau de Mestre em Ciência da Computação.

ORIENTADOR: NIVIO ZIVIANI

Belo Horizonte

Março de 2013

AÉCIO SOLANO RODRIGUES SANTOS

LEARNING TO SCHEDULE WEB PAGE UPDATES USING GENETIC PROGRAMMING

Dissertation presented to the Graduate Program in Computer Science of the Universidade Federal de Minas Gerais in partial fulfillment of the requirements for the degree of Master in Computer Science.

ADVISOR: NIVIO ZIVIANI

Belo Horizonte

March 2013

Santos, Aécio Solano Rodrigues

S237c Learning to Schedule Web Page Updates Using Genetic Programming / Aécio Solano Rodrigues Santos. — Belo Horizonte, 2013
xvi, 44 f. : il. ; 29cm

Dissertação (mestrado) — Universidade Federal de Minas Gerais. Departamento de Ciência da Computação.
Orientador: Nivio Ziviani

1. Incremental Web Crawling – thesis. 2. Scheduling Policies – thesis. 3. Genetic Programming – thesis. I.Orientador. II. Título.

CDU 519.6*73 (043)

UNIVERSIDADE FEDERAL DE MINAS GERAIS
INSTITUTO DE CIÊNCIAS EXATAS
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

FOLHA DE APROVAÇÃO

Escalonamento de atualizações de páginas web usando programação genética
(Learning to schedule web page updates using genetic programing)

AÉCIO SOLANO RODRIGUES SANTOS

Dissertação defendida e aprovada pela banca examinadora constituída pelos Senhores:

PROF. NIVIO ZIVIANI - Orientador
Departamento de Ciência da Computação - UFMG

PROFA. ANA CAROLINA BRANDÃO SALGADO
Centro de Informática - UFPE

PROF. EDLENO SILVA DE MOURA
Departamento de Ciência da Computação - UFAM

PROF. WAGNER MEIRA JÚNIOR
Departamento de Ciência da Computação - UFMG

Belo Horizonte, 11 de março de 2013.

Acknowledgments

Do you need somebody to get by? I do. This work was only possible with a "little help" from several people who I want to thank now.

First and foremost, I would like to thank my parents and my brother, which are the ones behind my will to pursue this goal. They are the ones from whom I most learned throughout my life and were always there to support me in whatever difficulties I have found.

I would like to thank also my advisors and professors from both Universidade Federal de Minas Gerais (UFMG) and Universidade Federal do Amazonas (UFAM), my friends from LATIN (Laboratório para Tratamento da Informação) from whom I learned a lot in the past two years. The opportunity of meeting these people and get some of their knowledge and experience made it worth these two years away from home, family and old friends.

Finally, I want to thank all the people that I never met, and probably will never meet in person, but were very important to this work. They range from people working to build free and open source software to scientists from all around the world who contributed to make this work possible.

To all these people, for their valuable help, time and inspiration: thank you so much. I already learned a lot from you and hope to keep on.

Abstract

One of the main challenges endured when designing a scheduling policy regarding freshness is to estimate the likelihood of a previously crawled web page being modified on the web, so that the scheduler can use this estimation to determine the order in which those pages should be visited. A good estimation of which pages have more chance of being modified allows the system to reduce the overall cost of monitoring its crawled web pages for keeping updated versions. In this work we present a novel approach that uses machine learning to generate score functions that produce accurate rankings of pages regarding their probability of being modified on the Web when compared to their previously crawled versions. We propose a flexible framework that uses Genetic Programming to evolve score functions to estimate the likelihood that a web page has been modified. We present a thorough experimental evaluation of the benefits of using the framework over five state-of-the-art baselines. Considering the Change Ratio metric, the values produced by our best evolved function show an improvement from 0.52 to 0.71 on average over the baselines.

Keywords: Incremental Web Crawling, Scheduling Policies, Genetic Programming.

Resumo

Um dos principais desafios enfrentados durante o desenvolvimento de políticas de escalonamento para atualizações de páginas web é estimar a probabilidade de uma página que já foi coletada previamente ser modificada na Web. Esta informação pode ser usada pelo escalonador de um coletor de páginas web para determinar a ordem na qual as páginas devem ser recoletadas, permitindo ao sistema reduzir o custo total de monitoramento das páginas coletadas para mantê-las atualizadas. Nesta dissertação é apresentada uma nova abordagem que usa aprendizado de máquina para gerar funções de *score* que produzem listas ordenadas de páginas com relação a probabilidade de terem sido modificadas na Web quando comparado com a última versão coletada. É proposto um arcabouço flexível que usa Programação Genética para evoluir funções que estimam a probabilidade de a página ter sido modificada. É apresentado ainda uma avaliação experimental dos benefícios de usar o arcabouço proposto em relação a cinco abordagens estado-da-arte. Considerando a métrica *Change Ratio*, os valores produzidos pela melhor função gerada pelo arcabouço proposto mostram uma melhora de 0.52 para 0.71, em média, em relação aos *baselines*.

Palavras-chave: Coleta Incremental de Páginas Web, Políticas de Escalonamento, Programação Genética.

Contents

Chapter 1

Introduction

1.1 Motivation

Search engines are systems used daily by users to find information on the Web. The three main components of a web search engine are: *web crawler*, *indexer*, and *query processor*. The web crawler is responsible for discovering and downloading new web pages, and maintaining a local repository with copies of the pages. The *indexer* builds a data structure called *inverted index* which stores the text of the downloaded pages in a compact format that allows for efficient search. The *query processor* is the component which receives the information needs of the user (usually as keywords) and sorts the documents by relevance according to an information retrieval model.

The development of a large scale web crawler presents several challenges related to the complexity of the algorithms and data structures needed to manage the amount of data available on the Web. The basic algorithm of a web crawler works in cycles, as described in what follows next. First, an initial set of URLs (known as *seeds*) is added to the set $\mathcal{S}$ of URLs scheduled to be downloaded. After that, the following steps are repeated until the objectives of the web crawler are reached:

1. The set $\mathcal{S}$ of scheduled URLs is downloaded;
2. New URLs are extracted from the downloaded pages and added to the local repository;
3. A new set $\mathcal{S}$ is chosen from the local repository to be downloaded in the next download cycle.

The step 3 is called *scheduling*. A *scheduling policy* is the algorithm that controls how the selection of pages is done. According to the desired objectives, different

scheduling policies can be used. For instance, a web crawler may want to achieve the following objectives:

- Coverage – The crawler must be able to download a large fraction of the pages available on the Web. A large coverage is desired because if a page is not downloaded, it will not be indexed and, hence, will not be shown in the search results.

- Importance – There is an unbounded number of web pages available in the web, but not all of them have useful content. Even if they had high quality content, no crawler could download all available pages because the resources are limited. Thus, a crawler should stop downloading pages at some moment and the pages considered more important should be downloaded first (Baeza-Yates et al., 2005; Cho and Schonfeld, 2007).

- Freshness – Due to the very dynamic nature of the Web, the local copies of the pages become outdated compared to the live copy on the Web. Maintain the local copies updated is important because if the pages remain outdated, the search results would show pages that do not exist anymore or that have different content compared to when it was downloaded. Web crawlers usually have access to a limited bandwidth and perform a complete scan for outdated pages would take so long that the repository would probably be full of outdated versions of pages even before the scan finishes. Thus, web crawlers must be able to predict what pages are more likely to change on the Web in order to update the local copies more efficiently.

- User Experience – A web search engine should avoid showing pages that do not exist anymore, or which are irrelevant to the user. On the other side, it is not worth update a web page if it does not improve search quality. Then, it may be desirable that a web crawler gives more priority to update pages that are more likely to be shown in the search results, so the probability of a user see an stale page in the search results is decreased (Pandey and Olston, 2005; Wolf et al., 2002).

- Discoverability – At every moment new pages are being created on the Web (Dasgupta et al., 2007). A web crawler must be able to discover and download these pages the fastest it can, so users searching for information available in recently created pages can find the information they want.

As the resources are limited, some of these objectives may be somewhat conflicting. For example, high freshness can be obtained by revisiting a large number of pages very often. However, the resources used to update a very large number of pages could be used to download novel pages, and hence, increase the coverage. Thus, it is necessary a scheduling policy that is able to balance the objectives to ensure proper freshness and at same time achieve a good coverage. This work contributes to the efforts for building good scheduling policies for web crawlers.

Another motivation is the ongoing work of development of a web search engine in the Information Retrieval research line of the Nation Institute of Science and Technology for the Web (InWeb). Some key algorithms for this crawler were already proposed in prior work, such as the algorithm for the verification of URL uniqueness (Henrique, 2011; Henrique et al., 2011) and algorithms for detection of web site replicas (Guidolini, 2011). The algorithm proposed in this work will be integrated to the scheduler of the InWeb crawler.

1.2 Objectives

The objective of this work is the development of good scheduling policies for web crawlers. We propose the use of a machine learning based framework to automatically build scheduling policies that optimizes a given objective. More specifically, we propose the use of Genetic Programming (GP) to learn functions that provide a score to each page regarding the given objective. Once the score function is learned, it can be applied efficiently to select the top-k pages to be downloaded in the next download cycle. In this work, we propose the application of the proposed framework to the task of scheduling web page updates with the objective of freshness maximization.

The main specific objectives of this work are:

- Propose the direct use of machine learning techniques to build scheduling policies for web crawler schedulers;
- Automatically build good policies for scheduling web page updates using the proposed machine learning based framework.
- Validate the hypothesis that GP provides a way to automatically build scheduling policies for web crawlers better than the state-of-the-art methods to schedule web page updates.

1.3 Contributions

The main contribution of this work is the proposal of a novel machine learning based framework to automatically build functions that can be used by web crawler schedulers. Machine learning techniques were already used in tasks related to scheduling in prior work, but none of them used machine learning to directly build functions to define the order that the pages will be downloaded as we do here.

We can cite as specific contributions of this work:

1. The proposal of a novel framework to build score functions that optimize a given objective and can be used in web crawler schedulers.
2. Description of how the proposed framework can be used to learn functions to schedule web page downloads for freshness maximization.
3. A thorough experimental evaluation of the the proposed framework applied to the freshness maximization problem using a dataset crawled from the Brazilian Web.
4. Description of how this framework can be integrated to a large scale crawler architecture such as the one described by Henrique et al. (2011).

1.4 Organization

The remainder of this work is organized as follows. Chapter 2 shows the basic concepts and related work regarding web crawlers and the Web Crawl Ordering problem. Chapter 3 presents the proposed framework, shows how to use it to schedule web page updates, and explains how the proposed framework can be integrated to a large scale web crawler. Chapter 4 shows an experimental evaluation of the framework. Finally, Chapter 5 shows the conclusions and future work.

Chapter 2

Basic Concepts and Related Work

In this chapter, we start by discussing basic concepts related to the architecture of web crawlers (Section 2.1). After that, we present the Web Crawl Ordering problem (Section 2.2) and prior work related to it, which are basically strategies to order web page downloads with the following objectives:

- acquire pages of high quality earlier in the crawl (Section 2.3);
- freshness maximization and discovery of novel pages (Section 2.4).

Finally, we present basic concepts of Genetic Programming (Section 2.5), which is the machine learning technique we use to build the scheduling functions in this work.

2.1 Web Crawler Architecture

In this section, we present a high-level description of the crawler architecture we adopt when developing our framework for scheduling web page downloads. The crawler architecture, further discussed in Henrique et al. (2011), has four main components: *fetcher*, *extractor of URLs*, *verifier of uniqueness* and *scheduler*. While our results can be applied to other crawler architectures, this architecture plays here the role of a sample, useful to give the reader a context about the problem we address. Figure 2.1 illustrates the crawl cycle involving the four components. The fetcher is the component that sees the Web. In step 1, the fetcher receives from the scheduler a set of URLs to download web pages. In step 2, the extractor of URLs parses each downloaded page and obtains a set of new URLs. In step 3, the uniqueness verifier checks each URL against the repository of unique URLs. In step 4, the scheduler chooses a new set of URLs to be sent to the fetcher, thus finishing one crawl cycle.

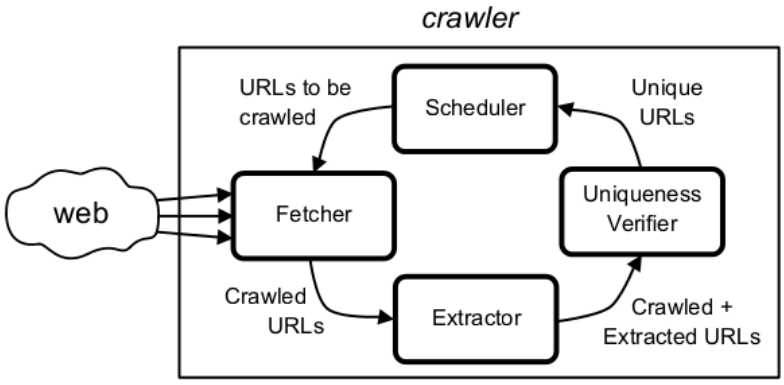

Figure 2.1. Web page crawling cycle.

Considering cycle i, the *fetcher* locates and downloads web pages. It receives from the *scheduler* a set of candidate URLs to be crawled and returns a set of URLs actually downloaded. The set of candidate URLs is determined by the amount of memory space available to the uniqueness verifier. The downloaded pages are stored on disk for efficient access and retrieval. The interval between two accesses to the same server must follow politeness rules, which might cause a significant slowdown in the whole process. In this work, we will not address these issues but rather concentrate on the algorithm for choosing the pages that will be downloaded at each cycle i. In the next section we discuss this problem in details.

2.2 Web Crawl Ordering Problem

Aside the politeness restrictions that a crawler must follow, the order of the downloads a crawler performs is free. The crawl order is a very important issue because it is not possible to download all pages available on the Web. To select a good crawling order there are two main issues: *coverage*, the fraction of desired pages that the crawler downloads successfully; and *freshness*, the degree to which the downloaded pages remain up-to-date, relative to the current live web copies. As the amount of crawling resources is finite, there is a trade-off between coverage and freshness. There is no consensus about the best way to balance the two. Olston and Najork (2010) argue that balancing the two objectives should be left as a business decision, i.e., it is a question of preferring broad coverage of content that may be somewhat out-of-date, or narrower coverage with fresher content.

Most work on Crawl Ordering Problem assume a simplified model, in which at every point in time the crawler has already acquired some content and must plan the

order in which the pages will be downloaded in the future. The order that the pages will be downloaded is determined by a *scheduling policy*, which relies on data already acquired by the crawler and other information sources to decide the future crawl order.

In this scenario, there are two approaches for managing downloads:

- Batch Crawling – The planned crawl ordering does not contains any duplicated URL, i.e. every page is downloaded only once. To acquire updated versions of a page, the entire crawl must be restarted and all pages must be downloaded again.

- Incremental Crawling – A URL may appear several times in the planned crawl, i.e. whenever an updated version of a page is needed, it can be downloaded again. Conceptually, the crawl is a continuous process that never ends.

Batch Crawling is usually used when only a snapshot of the Web is desired. When up-to-date versions of the pages are needed, Incremental Crawling is better because it has the advantage that it allows re-visitation of the pages at different crawling rates. It is believed that most commercial crawlers perform incremental crawling (Olston and Najork, 2010). In Section 2.3, we discuss previous work in Batch Crawl Ordering in details, whereas in Section 2.4 we discuss about Incremental Crawl Ordering.

2.3 Batch Crawling

2.3.1 Comprehensive Web Crawling

As mentioned earlier, the Web may be considered infinite due to its rate of growth and mainly because of the dynamic generated content. To illustrate this, we can think in a web page that displays a calendar with the current day of the month. This page may contain a link to another page that shows the next month. These pages create a infinite chain of pages with very similar (and possible useless) content. Thus, it is very important to web crawlers have means to acquire content of high quality earlier in the crawl.

A common metric used to measure the quality of the content acquired by a web crawler at any discrete time point t (assuming the crawler has a fixed crawl rate) is the *Weighted Coverage* (WC), defined as:

$$WC(t) = \sum_{p \in \mathcal{C}(t)} w(p)$$

where t denotes time since the crawl began, $\mathcal{C}(t)$ denotes the set of pages downloaded up to time t, and $w(p)$ denotes a numeric weight associated with the page p. The function $w(p)$ must reflect the importance of the page regarding the crawling objective.

Figure 2.2 illustrates a common way to compare different crawl policies using the Weighted Coverage metric. In the figure, we can see the results of four scheduling policies. *Omniscient* shows the results of a hypothetical policy which knows a priori the weights $w(p)$ of all pages and produces a perfect crawl ordering. It is a theoretic upper-bound curve. *Random* shows a basic policy which is linear in t. This is a baseline upon which all other policies should try to improve. A shows a policy which yields better results earlier in the crawl, whereas B shows a policy which performs better towards the end. The choice between the two depends on how much time the crawl will take.

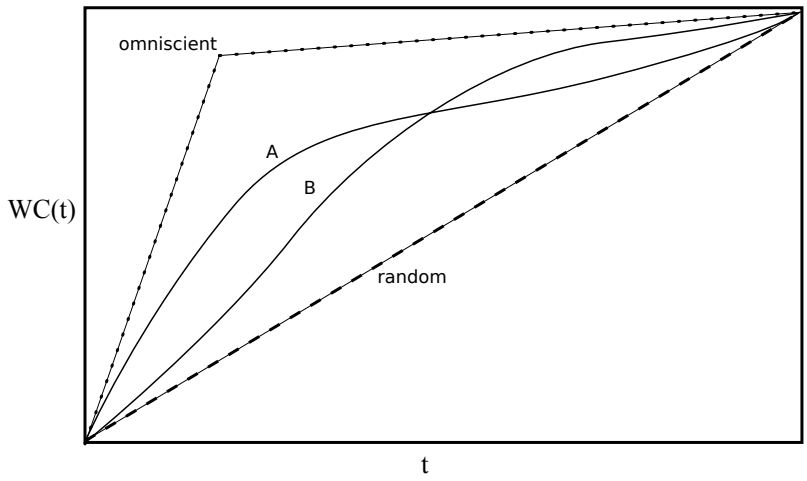

Figure 2.2. Weighted Coverage as a function of the elapsed time t since the beginning of the crawl (Figure based on Olston and Najork (2010)).

When the purpose of the crawl is to acquire content for a web search engine, a widely used metric to estimate the quality $w(p)$ of a web page is the PageRank (Page et al., 1998). Fetterly et al. (2009) evaluated the impact of crawl policies on web search effectiveness by measuring the maximum potential NDCG (Normalized Discounted Cumulative Gain) (Järvelin and Kekäläinen, 2002) that is achievable using a particular crawl policy. They showed that metrics such as in-degree, trans-domain in-degree and PageRank all allow better retrieval effectiveness than a simple breadth-first crawl[1]. They found also that PageRank is the most reliable and effective method of the three. In the next subsection, we present the scheduling policies previously proposed in the literature to acquire important pages earlier in the crawl.

[1]See Section 2.3.2 for a detailed explanation of this policies.

2.3.2 Scheduling Policies for Batch Crawling

Cho et al. (1998) were the first to study scheduling policies for Web Crawlers. They performed a crawl simulation using a collection of 180 thousand web pages crawled from the domain `stanford.edu`. In their study, they compared three scheduling policies:

- *Breadth-first* – which is a simple strategy that downloads pages in the same order they are discovered;
- *Backlink-count* – which gives more importance to a page p proportionally to the number of links found on the Web that point to p;
- *Partial PageRank* – which performs partial PageRank calculations using the data acquired so far.

Experimental results show that the policy *Partial PageRank* was the best one (Cho et al., 1998).

Abiteboul et al. (2003) proposed a scheduling policy based on an algorithm called OPIC (Adaptive On-line Page Importance Computation), which computes an approximation of PageRank during the crawler operation. In OPIC, all pages start with equal "cash" values, which are propagated to the URLs extracted from the downloaded pages. OPIC reduces the high overhead of the traditional PageRank computation because it does not require to store the complete link matrix.

Castillo et al. (2004) and Baeza-Yates et al. (2005) performed a comprehensive comparison of several existing scheduling policies and proposed new ones based on historical information of previous crawls. Initially, they evaluated the policies *Breadth-first*, *Backlink-count*, *Partial PageRank*, OPIC, and a new strategy called *Larger sites first*, which gives priority to download pages of the sites with the largest number of known pages. They also considered the *Omniscient* theoretical policy which knows a priori the importance of all pages. In their crawl simulation they also considered the politeness constraints a crawler must obey. The results showed that although the Omniscient policy is good in the beginning, towards the end the performance is close to a random policy. This happens because this policy is too greedy and downloads all pages with high PageRank very early, reducing the number of web sites available in the final stages of the crawl. A reduced number of sites can cause a significant slowdown because the crawler must wait some time between subsequent accesses to the same site. The policies Partial PageRank and Backlink-count did not perform well in their experiments because they tend to get stuck in pages that are local optimal, and then fail to discover other pages. The best policies without historical information

were OPIC and Larger-sites-first. They note that the policy Larger-sites-first was able to acquire a large coverage earlier, uses few resources and is very simple to implement.

The policies with historical information use the PageRank value of the web pages downloaded previously to guide future web crawls. Note that some web pages may have not been downloaded in the previous web crawls, so they propose the following alternative methods for using historical information to prioritize novel pages:

- *Historical-pagerank-omniscient* – new pages receive a weight from a oracle which knows the PageRank of all web pages;
- *Historical-pagerank-random* – new pages receive a random numeric weight;
- *Historical-pagerank-zero* – new pages receive weight equal to zero;
- *Historical-pagerank-parent* – new pages receive the PageRank value of the web page that it was discovered.

In general, the scheduling policies that use historical information performed better than the ones which do not have such information. The policy *Historical-pagerank-omniscient* was the one which performed better. Surprisingly, the strategy *Historical-pagerank-random* also performed well. Finally, although the policy OPIC (which does not uses historical information) did not perform well in the beginning, it was able to outperform the policies with historical information (except *Historical-pagerank-omniscient*) towards the end of the crawl.

Cho and Schonfeld (2007) proposed a new policy for crawl ordering based on an alternative way of computing PageRank, which uses summation of path probabilities. Their algorithm, referred to as RankMass, gives a high priority to important pages such as previous approaches, so that the search engine can index the most important part of the Web first. In addition, RankMass provides a theoretical guarantee on how much of the "important" part of the Web it will download after crawling a certain number of pages. This guarantee may be used to decide when to stop downloading the Web.

More recently, Alam et al. (2012) proposed a slight modification to the RankMass algorithm to skip the propagation of path probabilities towards the pages that were already downloaded. Besides that, they proposed a set of new algorithms which incorporates other features such as partial link structure, inter-host links, page titles, and topic relevance into the RankMass algorithm. Their set of algorithms, referred to as *Fractional PageRank*, were able to outperform the original RankMass algorithm in their experiments.

All scheduling policies presented in this section do not have a direct connection with this work, once here we focus only in the use of GP for scheduling Web page updates. However, the framework we propose here can be easily adapted to build scheduling policies for batch crawling with the objective of acquire important pages early. For that, it suffices to change the objective function that is being optimized and the features used by the GP framework. This effort is not studied here and is left for future work.

2.4 Incremental Crawling

An incremental crawler must interleave first-time download of new web pages with the re-visitation of already downloaded pages to maintain freshness. At each step, the crawler must decide between two actions:

1. Download a new page – which improves coverage, and may supply links to novel pages (which by its turn, improves the estimations of page importance, relevance, ability of spam detection, and so on);

2. Re-download an old page – may improve freshness, detect removed pages, and supply links to new pages as in the first action.

As mentioned earlier, there is no consensus about the best way to balance the two actions. Most published work on crawling focuses either on coverage or on freshness. In Section 2.3, we already reviewed previous works that deal with coverage. In the remainder of this section, we review the efforts on freshness. We discuss freshness maximization and freshness models in Section 2.4.1. In Section 2.4.2, we discuss techniques previously proposed for the problem of re-visiting known URLs to acquire an updated version of their content. These techniques are used by incremental crawlers to detect web page changes that may affect the quality of search engine results. Finally, in Section 2.4.3, we present the metrics commonly used to evaluate such techniques.

2.4.1 Maximizing Freshness and Freshness Models

One common goal of previous works is to maximize the Weighted Freshness (WF) of the local repository of pages, defined in Olston and Najork (2010) as:

$$WF(t) = \sum_{p \in C(t)} w(p) \cdot f(p,t)$$

where $WF(t)$ denotes the weighted freshness of the repository at time t, $C(t)$ denotes the set of pages crawled up to time t, $w(p)$ denotes a numeric weight associated with page p and $f(p,t)$ is the freshness of page p at time t, measured in ways we discuss next.

Most works assume that the set C of crawled pages is fixed for each crawling cycle, i.e., C is static, so there is no dependence on t. Its also assumed that each page $p \in C$ exhibits a stationary stochastic pattern of content changes over time (see Section 2.1 for the explanation of a crawling cycle).

There are two known models for measuring the freshness of a page. The first is the *binary freshness model*, in which $f(t,p) \in \{0,1\}$. More specifically:

$$f(p,t) = \begin{cases} 1 & \text{if the copy of } p \text{ is identical to the live copy on the Web} \\ 0 & \text{otherwise.} \end{cases}$$

The second way of measuring freshness is the *continuous freshness model*, in which some pages may be "fresher" than others. Considering this model, Cho and Garcia-Molina (2003a) proposed a temporal freshness metric called *age*, in which $f(p,t) \propto -age(p,t)$, where

$$age(p,t) = \begin{cases} 0 & \text{if the local copy of } p \text{ is identical to the live copy on the Web} \\ a & \text{otherwise} \end{cases}$$

where a denotes the amount of time the local copy has diverged from the copy on the Web. The intuition behind *age* is that the longer the local copy diverged from the live copy, the more their content tend to differ. Also considering the *continuous freshness model*, Olston and Pandey (2008) proposed a way of measure the freshness of web pages based directly on content. Instead of using the *age* of the page, they divided a page in a set content fragments and measured the fraction of fragments in common between the local copy and the live copy.

There is no consensus about what is the better freshness model in the literature. Some works (Cho and Garcia-Molina, 2000; Cho and Ntoulas, 2002; Tan and Mitra, 2010) adopt the binary model, while others adopt the continuous model (Olston and Pandey, 2008). In this work, we adopt the binary model and consider that all pages are equally important, i.e. $w(p) = 1$ for all pages.

2.4.2 Refresh Policies for Incremental Crawlers

Probabilistic models have been proposed to approximate the history of changes of web pages and to predict their changes in the future. Coffman et al. (1998) postulated a Poisson model of web page change. For each page p, the occurrence of change events is governed by a Poisson process with parameter λ_p, which means that changes occur randomly and independently, with an average of λ_p changes per time unit. In practice, the independence assumption may not strictly holds, but it has been shown that the Poison model provides a good approximation of the reality.

Cho and Garcia-Molina (2003b) studied how one can effectively estimate the change frequency of elements that are updated autonomously and independently. They identified various scenarios and proposed estimators for each of them. In the case of a web crawler, in which the information about change of pages is incomplete (i.e., a crawler knows whether a page has changed between accesses, but not the number of times it changed), they proposed the following estimator:

$$\lambda_p = -\log\left(\frac{n - X + 0.5}{n + 0.5}\right) \tag{2.1}$$

where n is the number of visits and X is the number of times that page p changed in the n visits. This estimator is an improved version of the intuitive frequency estimator $\frac{X}{T}$ (ratio of detected changes X in the monitoring period T), for the case of incomplete information about changes. Using real data, they showed that a web crawler can achieve improvement in freshness by setting its refresh policy to visit pages proportionally more often based on this improved estimator.

The strategy used by the crawler WebFountain (Edwards et al., 2001) does not assume any change model a priori. Its strategy categorizes pages adaptively into buckets based on their recent history of changes. Thus, it is not necessary to build a change model for each page explicitly, but only a model for each bucket.

Wolf et al. (2002) studied the problem considering a non-uniform change model, in which the change probability distribution is known. They proposed that the objective of a crawler must be avoid showing stale pages to the user in the search results. For that, the crawler must revisit preferentially frequently clicked pages in order to minimize the "embarrassment" incidents in which a search result shows a stale page.

Pandey and Olston (2005) argued that minimizing the embarrassment can not guarantee a good search experience because, even if the search results generates no embarrassment, the low quality of the returned web pages can still substantially degrade the user experience. In their work, they proposed the *user-centric* approach that is

based on the impact that an update may cause in the average quality of the local repository. The rationale for this approach is that, if the change does not affect the search engine results, there is no need to perform the update. The average quality can be measured as follows:

$$Avg.Quality \propto \sum_{q} \left(freq_q \cdot \sum_{p_i \in U} (V(p_i) \cdot (rel(q, p_i))) \right) \tag{2.2}$$

where $V(p_i)$ is the likelihood of viewing page p_i and is computed using a scoring function over a query q and the cached pages of the local repository; $rel(q, p_i)$ is the relevance computed using the same score function over q and the live copy of p_i; $freq_q$ is the frequency of query q. All these values can be obtained using the user logs of search engines and the downloaded pages.

Olston and Pandey (2008) introduced an approach that takes into account the longevity of the content. For example, the "today's blog entry", which remains for a long time in the page, is more important than the "quote of the day" fragment, which is substituted every day. Under this view, the objective of a crawler is to minimize the amount of incorrect content in the local repository. The optimal resource allocation policy proposed in Olston and Pandey (2008) differentiates between long-lived and ephemeral content, as well as frequently and infrequently changing pages.

Cho and Ntoulas (2002) proposed a sampling-based algorithm to detect web page changes. In their approach, first a sample of pages of the web site is downloaded. Then, the number of pages that changed is used to decide if the entire web site should be updated. Their sampling is at the level of a web site, which may not be a good granularity for grouping pages for updates, because the pages of a web site may have different change patterns (Fetterly et al., 2003).

Tan and Mitra (2010) proposed a clustering-based method to solve the aforementioned problem. In their approach, first the pages are grouped into k clusters of pages with similar change behavior using several features. Then, the clusters are sorted based on the mean change frequency of a representative cluster's sample. Finally, the algorithm proceeds downloading a sample of pages from the clusters to check if the pages have changed since their last download. If a significant number of pages in a cluster have changed, the remaining pages of the cluster are downloaded. The clusters are checked starting from the highest-ranked cluster to the lowest, or until the resources are exhausted. They proposed four different ways to calculate the weights associated with a change in each of the downloaded cycles, which we consider as baselines to compare with our GP approach.

Our work differs from Tan and Mitra (2010) in the sense that our approach is not sampling based, but rather uses machine learning to directly build a score function that allows the scheduling of web page updates. Once the score function has been learned, which is performed off-line, it can be applied on-line efficiently, thus allowing large scale web crawling using the architecture explained in Section 2.1. We notice that the ranking functions obtained by our GP approach may be used to sort the clusters in the clustering-based method, an effort that is left for future work.

Barbosa et al. (2005) were the first to exploit the use of static features extracted from the content of the pages to predict its change behavior. Based on this idea, Tan and Mitra (2010) proposed the use of new dynamic features, and other features extracted from the web link structure and web search logs to group pages with similar change behavior. More recently, Radinsky and Bennett (2013) went a step further and proposed a web page change prediction framework that uses not only features from the web page's content, but also the degree and relationship among the page's observed changes, the relatedness to other pages and the similarity in the kinds of changes they experienced. In this work, we do not exploit such features yet, as our interest is to assess the potential benefits of employing GP to build the scheduling functions. To the best of our knowledge, no previous work has exploited GP for that task. Thus, we here focus only on a basic set of features related to a single source of information: whether the page changed or not during each cycle. Nevertheless, given the flexibility of GP, our approach can be easily extended to include those other features in the future.

2.4.3 Evaluation Metrics for Incremental Crawlers

A widely used evaluation metric for incremental crawlers is the *Change Ratio*, which was proposed in Douglis et al. (1997). It can be used to measure the ability of the scheduling policy to detect web page changes. It can be defined as:

$$C_i = \frac{D_i^c}{D_i} \tag{2.3}$$

where D_i^c is the number of downloaded and changed web pages, and D_i is the total number of downloaded web pages, in the i^{th} download cycle. The intuition behind Change Ratio is that as higher is the concentration of changed pages in the set of scheduled URLs, as better is the scheduling. Notice that for defining Change Ratio it is necessary to determine the amount of pages to be downloaded in each download cycle.

In practice, some pages may be more important than others. The Change Ratio

metric does not capture this fact. To solve this probem, Cho and Ntoulas (2002) proposed a *Weighted Change Ratio*, which gives different weights $w(p)$ to the changed pages p. The weights $w(p)$ can represent different types of importance of pages, such as the PageRank of each page. The Weighted Change Ratio can be defined as follows:

$$C_i^w = \frac{1}{D_i} \sum_{p \in D_i} w(p) \cdot \mathrm{I}_1(p)$$

where $\mathrm{I}_1(p)$ is a an indicator function:

$$\mathrm{I}_1(p) = \begin{cases} 1 & \text{if } p \in D_i^c \\ 0 & \text{otherwise.} \end{cases}$$

Another way to measure the quality of a refresh policy is to evaluate the quality of the local repository with respect to the quality of users' experience when they use the search engine. The evaluation metrics that pursue this objective are referred to as *query-based metrics* in Tan and Mitra (2010). Wolf et al. (2002) proposed the *embarrassment level metric* which measures the probability that the user issues a query, click in a URL returned by the search engine, and finds that the web page is irrelevant to the query. Pandey and Olston (2005) proposed the *user-centric metric*, which measures the impact of the change in the average quality of the local repository after updating its pages. The average quality was measured using the equation 2.2 mentioned previously.

The query-based metrics measure the quality of the whole repository. Tan et al. (2007) proposed that the evaluation of incremental crawlers should consider only the web pages shown in the top of the ranking returned by the search engine. For that, they used the following metrics:

- Top-k Freshness – reflects the percentage of the web pages in the k first positions of the ranking returned by the search engine which are synchronized with the real copy on the Web.

- Top-k MAP (Mean Average Precision) – measures the average precision for the first k web pages of the ranking returned by the search engine. Note that this metric evaluates the relevance of the returned web pages instead of its freshness.

2.5 Genetic Programming

GP is a problem-solving technique based on the theory of evolution of species that adopts principles of biological inheritance and evolution of individuals in a popula-

tion (Koza, 1992). Given an optimization problem with a large space of solutions, it searches for a near-optimal solution by combining evolutionary selection and genetic operations to create better performing individuals in subsequent generations.

GP evolves a number of candidate solutions, called *individuals*, represented in memory as tree structures with pre-defined maximum depth d. Every internal node of the tree is a function and every leaf node, known as terminal, represents either a variable or a constant. The maximum number of available nodes of an individual is determined by the depth of the tree, which is defined before the evolution process begins. An example of an individual represented by a tree structure is provided in Figure 2.3.

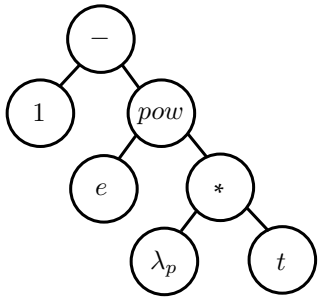

Figure 2.3. Tree structure representing the individual $1 - e^{\lambda_p t}$, which is the change probability function that will be presented in Equation 4.1.

The GP process starts with an *initial population* composed by N_p individuals. This initial population is generated randomly. Each individual is evaluated by a fitness function and is associated with a fitness value. This fitness function is defined by the problem where GP is applied and is used to guide the evolutionary process. For instance, it is used to select only those individuals that are closer to the desired goal or those that achieve better fitness results. The individuals will evolve generation by generation through the *reproduction*, *crossover*, and *mutation* genetic operations.

The *reproduction* operation consists in simply reproducing an individual of a generation into the next. The *mutation* operator has the role of ensuring some diversity of individuals in the population. It works by replacing randomly chosen subtrees of the individuals by another randomly chosen tree. The *crossover* operation allows genetic content exchange between two other individuals, the *parents*, selected among the best individuals of the current generation. Then, a random subtree is selected from each parent and a new individual, the *offspring*, is formed by swapping the selected subtrees of the parents. The offspring is then used to form the next generation.

Regarding the operations aforementioned, the following parameters should be set: *reproduction rate*, the percentage of elements which are copied to the next generation,

chosen among the best individuals according to the fitness function; *crossover rate*, the percentage of elements which will be used to the crossover operation; *mutation rate*, the percentage of elements which can be affected by mutations; Moreover, the *maximum crossover depth*, which is the maximum depth of trees given as input to the crossover operation, should also be defined.

At the end of the evolutionary process, a new population is created to replace the current one. The process is repeated over many generations until the termination criterion has been satisfied. This criterion can be a pre-established maximum number of generations N_g or some additional problem-specific success measure to be reached (e.g., an intended value of fitness for a specific individual).

2.6 Summary of the Chapter

In this chapter, we discussed several works related to the architecture of web crawlers, development of scheduling policies for ordering page downloads, and the machine learning technique called Genetic Programming. These are basic concepts used to build our proposed framework, which is described in the next chapter.

Chapter 3

Genetic Programming for Incremental Crawling

In this chapter we present GP4C – Genetic Programming for Crawling, a framework for building scheduling policies for web crawlers. In Section 2.5, we already discussed the basic concepts related to Genetic Programming (GP). From Section 3.1 to Section 3.5 we discuss how GP is applied to our target problem. In Section 3.6, we discuss how the scheduling policies created using GP4C can be incorporated in a large scale web crawler architecture. Finally, in Section 3.7, we provide an analysis of the computational costs involved in the framework.

3.1 GP4C – Genetic Programming for Crawling

In this section, we discuss how GP can be applied to the problem of scheduling web page updates. Specifically, we use GP to derive score functions that capture the likelihood that a page has been modified. Pages with higher likelihood of having been modified should receive higher scores, and ultimately higher priority in the scheduling process.

Our GP process is adapted from the one presented in da Costa Carvalho et al. (2012), where GP is applied to learn how to mix a set of sources of relevance evidence in a search engine at indexing time. The main steps of our method are described in Algorithm 1. It is an iterative process with two phases: *training* (lines 1–8) and *validation* (lines 9–11).

We create our training and validation sets considering a general updating scenario, in which an initial set of pages is used for training and the learned functions are validated using a distinct set of pages. This scenario is close to the one found in large crawling tasks, such as when performing a crawling to a world wide search engine,

where a scheduling policy must generalize well for web pages unseen in the training phase.

As shown in Algorithm 1, GP4C starts with the creation of an initial random population of individuals (line 1) that evolves generation by generation using genetic operations (line 8). The process continues until the number of generations of the evolutionary process reaches a maximum value given as input. Recall that, in the training phase, a fitness function is applied to evaluate all individuals of each generation (lines 5–6), so that best individuals are more likely to be selected to continue evolving than inferior individuals (line 8). Furthermore, the N_b best individuals found across all generations are stored (line 7) to be evaluated using the validation set.

After the last generation is created, the validation phase (lines 9–11) is applied to avoid selecting individuals that work well in the training set but do not generalize for different pages (a problem known as *over-fitting*). In this phase, the fitness function is also used, but this time over the validation set. Individuals that perform the best in this phase are selected as the final scheduling solutions (line 12).

Algorithm 1: Genetic Programming for Crawling (GP4C)

input : $\mathcal{T}$: a training set of pages crawled in a given period;
$\mathcal{V}$: a validation set of pages crawled in a given period;
N_g: the number of generations;
N_b: the number of best individuals maintained for validation;

1 $\mathcal{P} \leftarrow$ Initial random population of individuals;
2 $\mathcal{B}_t \leftarrow \varnothing$;
3 **foreach** *generation* g *of* N_g *generations* **do**
4 $\quad \mathcal{F}_t \leftarrow \varnothing$;
5 $\quad$ **foreach** *individual* $i \in \mathcal{P}$ **do**
6 $\quad\quad \mathcal{F}_t \leftarrow \mathcal{F}_t \cup \{g, i, \texttt{fitness}(i, \mathcal{T})\}$
7 $\quad \mathcal{B}_t \leftarrow \texttt{getBestIndividuals}(N_b, \mathcal{B}_t \cup \mathcal{F}_t)$;
8 $\quad \mathcal{P} \leftarrow \texttt{applyGeneticOperations}(\mathcal{P}, \mathcal{F}_t, g)$;
9 $\mathcal{B}_v \leftarrow \varnothing$;
10 **foreach** *individual* $i \in \mathcal{B}_t$ **do**
11 $\quad \mathcal{B}_v \leftarrow \mathcal{B}_v \cup \{i, \texttt{fitness}(i, \mathcal{V}))\}$
12 BestIndividual $\leftarrow \texttt{applySelectionMethod}(\mathcal{B}_t, \mathcal{B}_v)$;

3.2 Individuals, Terminals and Functions

In the case of GP4C, each individual represents a function that assigns a score to each page. Such score combines information useful for estimating the likelihood of a given

page being updated in a period of time, taking into account information such as its behavior in previous downloads.

An *individual* is represented by terminals (leaves) and functions (inner nodes), organized in a binary tree structure, as illustrated in Figure 2.3. Terminals contain information (features) obtained from the pages that may help in the task of characterizing their updating behavior and thus can be useful as parameters to compose the final score function.

In our GP4C approach we considered only three basic *terminals* that contain information about the pages:

1. n: Number of times that the page was visited;

2. X: Number of times that the page changed in n visits;

3. t: Number of cycles since the page was last visited.

In addition to these terminals, we also used *constant values*. Each constant is a different terminal. They are: $0.001; 0.01; 0.1; 0.5; 1; e; 10; 100; 1000$.

As *functions* in the inner nodes, we use addition ($+$), subtraction ($-$), multiplication ($*$), division ($/$), logarithm (log), exponentiation (pow), and the exponential function (exp). Specifically for the functions division and logarithm, we used protected versions to avoid the computation of log and division by zero. For division, if the denominator is zero (or very close to zero), we return a very large number with the same signal. For logarithm, we consider that $\log 0$ returns 0.

All *terminals* and *functions* we use in GP4C were extracted from functions found in the literature that we use as baselines. The rationale for using them is that GP can recombine these functions and terminals to create more effective functions than the ones proposed in the literature.

3.3 Genetic Operations

GP relies on a set of genetic operators to evolve the population across generations. These operators probabilistically combine genes of individuals based on its fitness value in order to create new candidate solutions for the problem. In GP4C, we use the genetic operators *reproduction*, *crossover* and *mutation*. The *reproduction* operator simply makes a copy of an individual to the next generation. We also employ *elitism*, which means that the best individual of a population is always reproduced in the

next generation. Regarding the *mutation* operation, we use two types available in the GPC++ class library[1]:

1. *Node replacement mutation* (also known as *point mutation*) – a node of the tree is chosen at random, and its value is swapped by another random value. Terminals are always replaced by other terminals and functions are always replaced by other functions with the same number of arguments. For instance, Figure 3.1 shows an individual that the terminal e was randomly chosen and swapped by the terminal n. Figure 3.2 shows a mutation in a function node, where the function exponentiation (*pow*) was swapped by the function division (/).

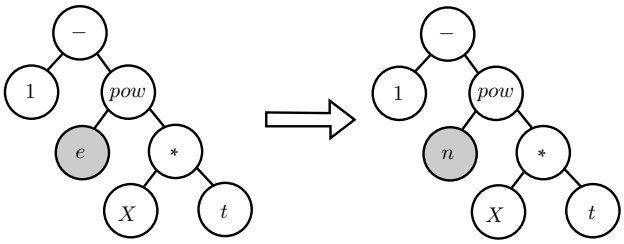

Figure 3.1. Example of node replacement mutation in a terminal node.

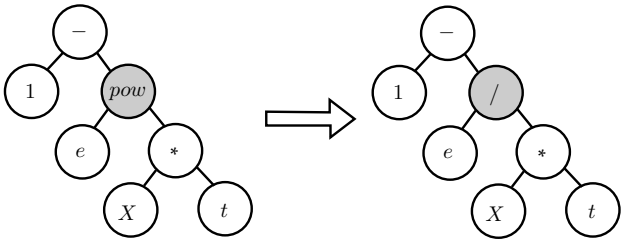

Figure 3.2. Example of node replacement mutation in a function node.

2. *Shrink mutation* – a function node is chosen at random and is replaced by a randomly chosen terminal, as shown in Figure 3.3. The shrink mutation causes a reduction in the size of the tree, which reduces its structural complexity and can make it easier to understand.

For *crossover* operation, two parent individuals are randomly selected among the top best individuals of the current generation. Then, a node in each parent is chosen at random. Finally, the subtrees located in the selected nodes are swapped, creating two new individuals called *offsprings*. Figure 3.4 shows an example of two individuals being combined using crossover.

[1]GPC++ refers to *node replacement mutation* as *swap mutation*. Here, we follow the same terminology of Poli et al. (2008) and use the term *node replacement*.

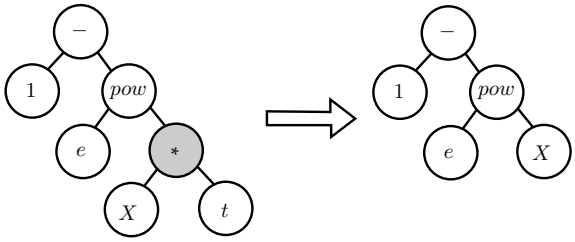

Figure 3.3. Example of shrink mutation.

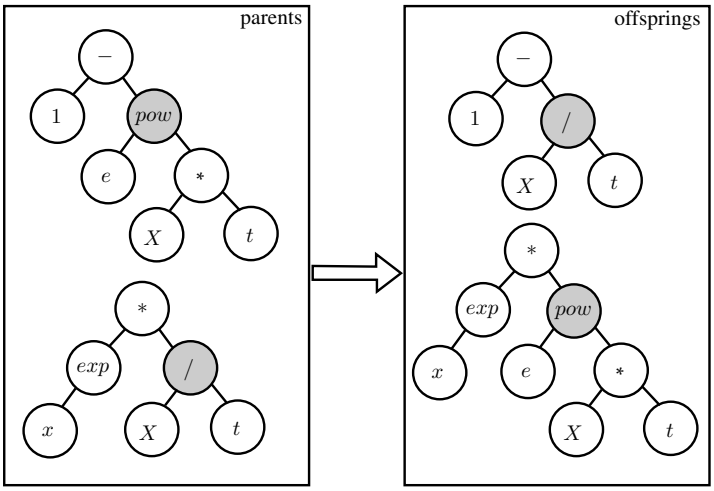

Figure 3.4. Example of crossover operation.

3.4 Fitness Function

In GP4C, the fitness function measures the quality of the ranking generated using a given individual for the whole training period. Algorithm 2 presents the computation of the fitness of an individual. To compute the fitness of an individual, we perform a crawl simulation. The *training set* is used to create an "oracle" which has information about the changes of all pages for the whole training period. Then, the oracle is used to simulate the downloads. Instead of performing a connection to the Web to download a page, the oracle is asked whether the page changed or not since the last time it was visited. As shown in Algorithm 2, the crawl simulation starts by creating a local repository of pages (line 1). Then, we must ensure that all pages of the repository are revisited some times in order to get basic information about its change behavior (lines 2–4). After that, the remaining download cycles available (line 5) in the training set are used to perform the crawl simulation (lines 6–14) to evaluate the performance of the individual. For that, we take the score it produces for each page (lines 9–10) in the training set of each given download cycle and generate a schedule by choosing the

pages with the highest scores (line 11) to be crawled in the next download cycle (lines 12–13). The quality of this ranking is then evaluated by the fitness function (line 14). We adopted the *Change Ratio* metric as fitness function, which means that the quality of the ranking is the ratio of pages that were scheduled to be downloaded and that really changed in a given download cycle[2]. Finally, the final fitness value is the average Change Ratio produced in all simulated download cycles (line 15).

Algorithm 2: Fitness Function Computation

input : f: a scheduling function represented by an individual;
$\mathcal{O}$: an oracle which knows when each page changed;
N_o: the number of download cycles that $\mathcal{O}$ is aware of changes;
N_w: the number of times a page must be visited to get basic information of change;
k: the max number of pages that can be downloaded using the available resources;

```
1  R ← A new local repository of pages;
2  foreach warm-up cycle w of N_w do
3      foreach page p of R do
4          updateChangeInformation(p, w, O);
5  N_sim ← N_o − N_w;
6  C ← ∅;
7  foreach download cycle c of N_sim do
8      W ← ∅;
9      foreach page p of R do
10         W ← W ∪ {c, p, score(p, c, f)};
11     S ← selectPagesWithHighestScores(R, W, k);
12     foreach page p ∈ S do
13         updateChangeInformation(p, c, O);
14     C ← C ∪ {c, evaluateScheduling(S, c, O)} ;
15 return average(C);
```

3.5 Selection of the Best Individuals

The selection of the best individual is accomplished by running the GP4C process with a set of distinct randomly selected seeds (da Costa Carvalho et al., 2012). The whole GP process depends on the selection of an initial random seed to produce its results. To reduce the possible risks of finding a low performance local best individual, we run

[2]See Section 2.4.3 for the definition of the Change Ratio metric

N processes with distinct random seeds, and pick the best individual among those generated by these N runs.

The best individual of a GP4C process is chosen using a selection method (line 12 of Algorithm 1). The first selection method we used choses the individual that performed better in the training phase. We refer to this approach as $GP4C_{Best}$. As proposed in de Almeida et al. (2007), we also consider two other selection strategies that are based on the average and the sum of the performances of each individual in both the training and validation sets, minus the standard deviation value of such performance when selecting best individuals. In this work, the standard deviation was computed using the Change Ratio produced in all download cycles of the training and validation sets. In de Almeida et al. (2007), the authors referred to these methods as Avg_σ and Sum_σ. The individual with the highest value of Sum_σ (or Avg_σ) is selected as the best. Here, we refer to GP4C using these selection strategies as $GP4C_{Sum}$ and $GP4C_{Avg}$. These specific selection strategies are used to avoid selecting an individual that perform well in the training set, but do not generalize well for unseen web pages. Besides that, these strategies are useful to produce more stable results when running a GP process.

3.6 Using GP4C in a Large Scale Web Crawler

Some web crawler architectures, such as the one described in Section 2.1, use a scoring function to choose the pages that will be visited in each download cycle. Then, the functions learned using GP4C are suitable for selecting the web pages that will be downloaded in each download cycle. The training phase of GP4C, where the scheduling function is learned, must be performed off-line. Once the function is learned, the score of each page can be computed efficiently in constant time during the crawler operation. To choose the pages, a sort can be performed to pick the k pages with the highest scores.

Large scale crawlers deal with billions of pages, so its unfeasible to perform random accesses to the repository to update the pages (as well as to verify uniqueness of the extracted links at each download cycle). To solve this problem, state-of-the-art algorithms (Henrique et al., 2011; Lee et al., 2009) accumulate the links extracted by the *Extractor* in a buffer, so the verification of uniqueness can be performed efficiently in batches in a single-pass in the repository. The terminals used in this work can be computed efficiently while performing the verification of unique URLs. During the uniqueness check, whenever the crawler finds that a downloaded page already exists in the repository, it can update its terminal values.

3.7 Computational Costs

The computational cost to apply the function during the crawler operation depends on the time to compute the terminal values of the learned function. As mentioned earlier, all terminals used in this work can be computed efficiently in constant time during the crawler operation. Thus, the cost to apply the function on-line is given by the time necessary to compute the score, plus the time to pick the pages with highest scores.

Now we turn to the running time of the training phase that is performed off-line. The most time consuming task in the GP4C process, shown in Algorithm 1, is the computation of the fitness of each individual (line 6), which is accomplished by the crawl simulation further detailed in Algorithm 2. The number of crawl simulations performed is determined by the number of generations N_g and the size of the population $|\mathcal{P}|$, plus the number of best individuals $|\mathcal{B}_t|$ maintained for the validation phase. Then, the total number of crawl simulations that must be performed in the worst case is given by:

$$N_g \times |\mathcal{P}| + |\mathcal{B}_t|$$

The values N_g and $|\mathcal{P}|$ are determined empirically, but usually values smaller than 500 are sufficient for convergence. $|\mathcal{B}_t|$ must also be a small number, otherwise bad performing individuals would be selected for the validation phase.

The cost of a crawl simulation is determined mainly by the number of pages and number of download cycles available in the training set, as shown in the following analysis of Algorithm 2. First, in line 1, a new repository of $|\mathcal{R}|$ pages is created, being $|\mathcal{R}|$ the number of pages available in the training set. In lines 2–4, $N_w \times |\mathcal{R}|$ downloads of pages are simulated. In practice, N_w must be a small number since it is unfeasible for a web crawler download all pages several times. Then, for each remaining cycle N_{sim}, the following steps must be performed. The score of $|\mathcal{R}|$ pages must be computed (lines 9–10). Since the individuals are represented using a *tree*, a traversal must be performed to compute the final score value, which give us a cost of $|\mathcal{R}| \times n$, being n the average size of the trees. In line 11, a sort with worst case cost of $|\mathcal{R}| \times \log |\mathcal{R}|$ must be performed to pick the pages with the highest scores. In lines 12–13, $|S|$ pages must be checked for change and have its change information updated. $|S|$ is equal to k, which is the number of pages that can be downloaded using the resources available. In practice, k is a fraction of the number of pages $|\mathcal{R}|$ of the repository. Finally, in line 15, the average of N_{sim} values must be computed. Thus, considering only most costly operations, the worst case performance of the Algorithm 2 is in the order of:

$$O(N_o \times |\mathcal{R}| \times \log |\mathcal{R}|)$$

If we consider the total cost of all crawl simulations in the GP4C process, it is in the order of:

$$O\left((N_g \times |\mathcal{P}| + |\mathcal{B}_t|) \times (N_o \times |\mathcal{R}| \times \log |\mathcal{R}|)\right)$$

Despite the total computational cost of the GP4C process be quite high, it can be performed in a reasonable time, as shown in the experiments in Chapter 4. Furthermore, each crawl simulation is data independent from each other. Thus, if more computing power is available, the computation of the fitness of each individual can be distributed into multiple processors.

3.8 Summary of the Chapter

In this chapter we presented GP4C, our Genetic Programming based framework for generating scheduling policies for web crawlers. We also explained how the framework can be used in a large scale web crawler architecture and provided an analysis of computational costs of the framework.

Chapter 4

Experimental Evaluation

In this chapter, we present an experimental evaluation of the GP4C framework. We evaluate the effectiveness of the scheduling policies generated by GP4C using a dataset crawled from the Brazilian Web. We compare GP4C with five functions found in the literature used to estimate the likelihood of a page being modified on the Web since its last visit. We also measured the running time of the GP4C process to show that it is a practical approach that can be used by large scale web crawler schedulers.

In Section 4.1, we present the baselines and in Section 4.2, we present the dataset used in our evaluation. In Section 4.3, we describe the experimental methodology and evaluation metric. In Sections 4.4 and 4.5 we present the GP parameters and setup used to run the experiments. Some representative results of the scheduling policies created by GP4C are presented in Section 4.6. Finally, we show the performance of GP4C in Section 4.7.

4.1 Baselines

To evaluate our framework, we compare the three proposed strategies, $GP4C_{Best}$, $GP4C_{Sum}$ and $GP4C_{Avg}$, with five baselines we called CG, NAD, SAD, AAD and GAD. The five baselines are described next.

The first baseline we consider, CG, is the change frequency estimator proposed in Cho and Garcia-Molina (2003b), which is defined as:

$$\mathrm{CG} = -\log\left(\frac{n - X + 0.5}{n + 0.5}\right)$$

where n is the number of visits and X is the number of times that page p changed in the n visits.

The other four functions we consider as baselines were proposed in Tan and Mitra (2010). In order to compute the change probability of the pages, they assume that each page p follows a Poisson process with parameter λ_p. Considering T the time that the next change will happen, the probability φ that the page will change in the interval $(0, t]$ is calculated by integrating the following probability density function:

$$\varphi = Pr\{T \leq t\} = \int_0^t f_p(t)dt = \int_0^t \lambda_p e^{-\lambda_p t} dt = 1 - e^{\lambda_p t} \tag{4.1}$$

As φ depends on the change frequency parameter λ_p and time t, we set t to be the number of cycles since the page was last downloaded and compute λ_p using the change history of the pages:

$$\lambda_p = \sum_{i=1}^{n} w_i \cdot \mathrm{I}_i(p),$$

where n is the number of times the page was downloaded so far, w_i is a weight associated with a change occurred in the i^{th} download of the page ($\sum_{i=1}^{n} w_i = 1$), and $\mathrm{I}_i(p)$ is an indicator function defined as:

$$\mathrm{I}_i(p) = \begin{cases} 1 & \text{if page } p \text{ changed in the } i^{th} \text{ download} \\ 0 & \text{otherwise.} \end{cases}$$

We computed the weights w_i using the same four adaptive settings proposed in Tan and Mitra (2010):

- NAD (*Nonadaptive*) – considers that all changes have the same importance, that is:
$$w_1 = w_2 = \cdots = w_n = \frac{1}{n}$$

- SAD (*Shortsighted adaptive*) – considers that only the last change is important, that is:
$$w_1 = w_2 = \cdots = w_{n-1} = 0, w_n = 1$$

- AAD (*Arithmetically adaptive*) – considers that the most recent changes have more importance and that the weight of the more old changes decrease slowly following an arithmetic progression:
$$w_i = \frac{i}{\sum_{i=1}^{n} i}$$

- GAD (*Geometrically adaptive*) – as the previous setting, considers that the most recent changes have greater importance. However, the importance decreases more quickly, following geometric progression:

$$w_i = \frac{2^{i-1}}{\sum_{i=1}^{n} 2^{i-1}}$$

Summarizing, the baselines NAD, SAD, AAD and GAD we consider are computed using the change probability as defined in Equation 4.1, using the four weight settings proposed in Tan and Mitra (2010).

We also consider two simpler approaches to build score functions, referred to as *Rand* and *Age*. In *Rand*, the scores are random numeric values, whereas in *Age*, they are equal to the time t since the page was last downloaded.

4.2 Dataset

Our evaluation was performed using a web page dataset collected from the Brazilian Web (`.br` domain). This dataset was gathered using the crawler presented in Henrique et al. (2011), whose architecture was described in Section 2.1. Table 4.1 summarizes the final dataset, which is referred to as BRDC'12 from now on[1]. It was collected between September and November 2012. The dataset consists of a fixed set of web pages, which were crawled on a daily basis during approximately two months.

Table 4.1. Overview of our dataset (after filtering errors).

BRDC'12	
Monitoring period	57 days
# web pages	417,048
# web sites	7,171
Min # web pages/site	1
Max # web pages/site	2,336
Average web pages/site	58.15
% downloads with errors	2.92

To build BRDC'12, we used as seeds approximately 15,000 URLs of the most popular Brazilian sites according to Alexa[2]. Only sites under the `.br` domain were considered as seeds. A breadth-first crawl from these seeds downloaded around 200

[1]BRDC'12 dataset is available on-line at http://homepages.dcc.ufmg.br/~aeciosantos/datasets/brdc12
[2]http://www.alexa.com/topsites/countries/BR.

million web pages, from where more URLs were further extracted. From these URLs, we then selected a set of 10,000 web sites using stratified random sampling, thus keeping the same distribution of the number of web pages per site of the complete dataset. Next, for each selected site, we chose the largest number of web pages that could be crawled in one day without violating politeness constraints. We selected, in total, 3,059,698 web pages, which were then daily monitored. The complete BRDC'12 has about 1 Tb of data.

During the monitoring periods, our crawler run from 0 AM to approximately 11 PM, recollecting each selected web page every day, which allowed us to determine when each page was modified. In order to detect changes in a page, we used the SimHash technique (Manku et al., 2007) to create a fingerprint of the plain text extracted from the pages. Accesses to web pages from the same site were equally spaced to avoid hitting a web site too often.

We did observe download errors during monitoring periods. Such errors might be due to, for instance, the page being removed, the web site's access permissions (robots.txt) being changed, or the download time reaching a predefined limit (30 seconds). We removed from the BRDC'12 collection all web pages with more than two errors, thus including only pages with fewer errors in our analysis.

Note that, in case of an error, we cannot tell whether the page changed on that particular day. Thus, we guess this information by analyzing the history of changes of that web page in the days that preceded the error. Specifically, let us say the download of a page p failed on day d. We then analyze the distribution of the number of days between successive changes of p in the first $d-1$ days, and use the most frequent period without change to determine whether we should consider that p changed on day d.

Note from Table 4.1 that the filtered dataset contains a number of web pages larger than previous work (Tan and Mitra, 2010). Note also that the errors that remain in the BRDC'12 collection after filtering represents only 2.92% of all downloads performed. Although these errors might somewhat impact the quantitative results of each method, we note that all considered approaches might be affected. Thus, the remaining errors should not significantly impact our conclusions.

4.3 Experimental Methodology

Our evaluation was performed using 5-fold cross validation. The dataset was divided into 5 equal-sized folds; 4 folds were further equally divided into *training set* and *validation set*, and the last fold was used as *test set*. The training set was used to

evolve the population of individuals in the GP4C process, whereas the *validation set* was used to choose the best individuals, as discussed in Section 3.5, particularly to compute Avg_σ and Sum_σ. The best individuals selected were evaluated using the *test set*.

In order to evaluate the score functions and compute fitness values we simulate a crawl as described in Algorithm 2. Our simulation starts with a warm-up period N_w=2 days, during which data of BRDC'12 is used to build basic statistics about each page. For each day following warm-up, we apply our proposed score function and each baseline to assign scores to each page. The download of the top-k pages with highest scores (i.e., most likely of having been changed) is then simulated by updating statistics of the page such as number of visits (i.e., downloads), number of changes, etc. Specifically, we use the collected data to determine whether a page changed or not since the last time it was visited.

Our main *evaluation metric* is Change Ratio (Equation 2.3). In the case of GP4C, the metric Change Ratio is used both as evaluation criterium and fitness function. When evaluating a scheduling using Change Ratio, it is necessary to determine the maximum number of web pages that can be crawled in each day, referred to as k in our experiments. The k value represents the resources available to crawler. Once the resources are limited, k must be a small fraction of the pages present in the repository. We here set k equal to 5% of the total number of web pages in the dataset, which is a value close to values used in previous work. For each day, we compute the Change Ratio metric using the top-k pages in the sorted list produced by each method. Notice that, whenever the actual number of pages changed in a day is smaller than k, no evaluated algorithm can reach a maximum Change Ratio. This particular detail may cause variations in the Change Ratio obtained by a function when comparing results in distinct days. This variation however does not affect the conclusions about the comparison of the relative performance of each method.

4.4 GP Parameters

All GP operations were implemented using the open-source class library GPC++[3]. Regarding parametrization of the GP framework, we experimented several values for each parameter and selected the best ones we found, as described next. We set N_p equal to 300 individuals, created using the ramped half-and-half method (Koza, 1992). Due to the stability of results, we set N_g equal to 50 generations as termination criterion.

[3]GPC++ is freely available at http://www0.cs.ucl.ac.uk/staff/W.Langdon/ftp/weinbenner/gp.html (accessed in February 22, 2013).

We adopted tournament selection of size 2 to select individuals to evolve and set the crossover, reproduction, shrink mutation, and node replacement mutation rates equal to 90%, 15%, 5% and 5%, respectively. We set the *maximum tree depth d* to 10 and the *maximum depth for crossover* to 9. During the evolution process we kept the $N_b = 50$ best individuals discovered through all generations to the validation phase. We run the GP4C process using 5 random seed values.

4.5 Setup

All experiments were executed in server machines PowerEdge R710 Dell with 64 Gb of RAM memory and 2 processors Intel Xeon X5680 of 3.33 GHz with 6 cores. Despite of the large amount of memory available, the GP4C process used less than 3% of available memory during the evolution process. The GP4C process is a CPU intensive task, consuming about 100% of processing power during evaluation of fitness of the individuals. The fitness evaluation was parallelized using threads to take advantage of the multiple processors available.

4.6 Results

We now discuss the results obtained by our GP4C framework and the baselines using the data in the BRDC'12 collection. We present a experiment using only a basic set of terminals, n, X and t (see Section 3.2), to show that our GP framework can derive good functions which are equal or better than the ones used as baselines.

We start by comparing the Change Ratio produced by the three strategies $GP4C_{Best}$, $GP4C_{Avg}$ and $GP4C_{Sum}$ (see Section 3.5 for description of the strategies), on each day. Figure 4.1 shows the average Change Ratio for each day computed across all 5 folds. We omit 95% confidence intervals to improve clarity. Note that all three curves are very close to each other. Indeed, we performed a pairwise t-test for each pair of methods finding that all three methods are statistically tied, with 95% confidence, for almost all days. The only exception is for $GP4C_{Best}$, which outperforms $GP4C_{Avg}$ in one day and $GP4C_{Sum}$ in two days. In this case, the fact that $GP4C_{Best}$ has good performance in both *training set* and *test set* suggests that the individuals generated by GP4C framework do not suffer of *over-fitting* and generalize well for unseen web pages. Thus, we chose $GP4C_{Best}$ to compare against the baselines.

When observing the results of the the experiments in detail, we realized that the functions generated by GP4C are quite stable when changing the set of pages where

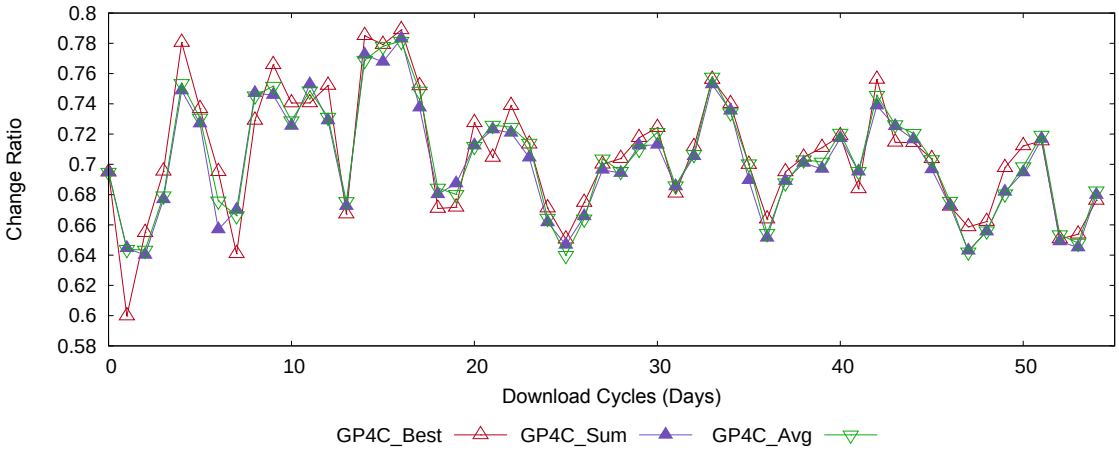

Figure 4.1. (Color online) Average Change Ratio for our three GP4C approaches computed across all folds of each download cycle

they are applied. Table 4.2 shows the Change Ratio obtained when applying the best functions to the training, validation and test sets. As we can see, the results obtained in the test set are pretty close to the ones obtained in the training and validation sets in all folds. These results indicate GP4C have produced quite stable and generic functions, which is one of the properties desired when applying machine learning solutions to any problem.

Table 4.2. Change Ratio of best functions found by $GP4C_{Best}$ in each fold

Fold	Train set	Validation set	Test set
1	0.709357	0.708086	0.703514
2	0.704912	0.698197	0.706889
3	0.718066	0.715886	0.711833
4	0.706658	0.708875	0.713094
5	0.696098	0.694009	0.693782

An interesting property of our GP4C framework is that it can also be used as a tool for better understanding the scheduling problem. For instance, we realized that quite simple functions provide results superior to the ones achieved by the baselines. Further, when analyzing the results, we also realized that the best ones followed the pattern giving more importance to the number of cycles since the page was last visited (t) and to the number of times it changed in last visits (X). We notice that the value of n does not impact much the final score in such functions. That is probably due to the fact that the period of crawling adopted for training was about only two months. In larger periods, the importance of n may increase, but experiments in larger periods are now unfeasible for us, since it requires a continuous crawling. We stress anyway the ability of our method to adapt its functions to the dataset given for training.

To show an example, we present an extremely simple, but also effective, function generated by our method: $t * X$.

While simple, this function resulted in final performance superior to most of the baselines, achieving average Change Ratio above 0.69. It was not the best function found by GP4C, but illustrates how the framework can be applied not only to derive good score functions, but also to give insights about the most important parameters.

Figure 4.2 shows the Change Ratio results of the best GP4C variation ($GP4C_{Best}$) and all considered baselines, namely NAD, SAD, AAD, GAD and CG. Once again, we show only average results to improve readability, but we draw our conclusions from statistical pairwise t-tests (with 95% confidence). We note that $GP4C_{Best}$ is statistically superior, with 95% confidence, to each of the baseline methods in most days, being tied to these methods in just a few days. In particular, we find that NAD, AAD, GAD and CG are the the most competitive baselines.

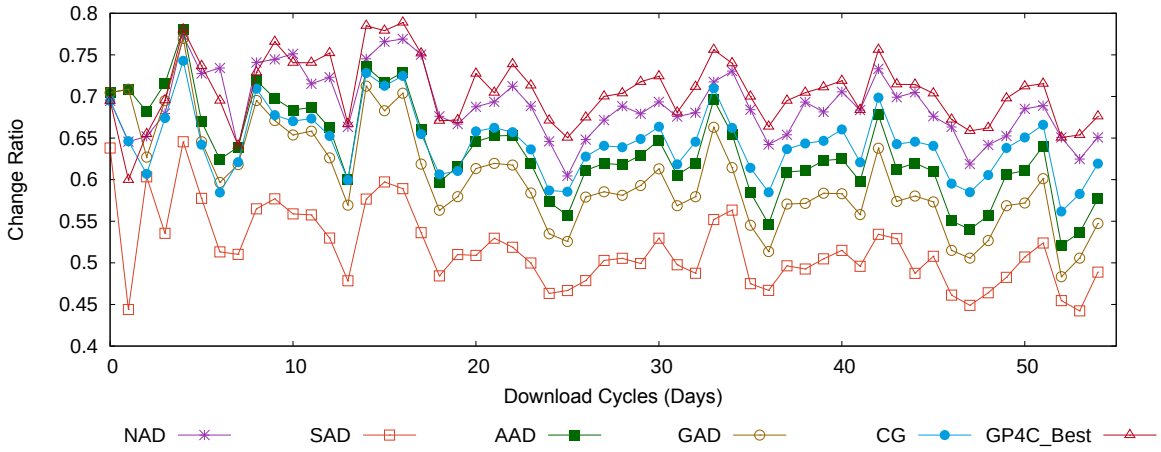

Figure 4.2. (Color online) Average Change Ratio for $GP4C_{Best}$ and the five baselines, computed across all folds of each download cycle

$GP4C_{Best}$ is statistically superior to NAD, AAD, GAD and CG in 22, 47, 49 and 50 of the simulated download cycles, respectively, being statistically tied with them in all other days. The only exception is for the three initial days of download cycles, when $GP4C_{Best}$ is statistically inferior to AAD in days 1 and 3 and to GAD in day 1. In other words, after the third day, $GP4C_{Best}$ is not outperformed by any of the baselines in any of the download cycles. This result corroborates the flexibility of our framework as it is able to produce results at least as good, if not better, than all five baselines.

Unlike the findings of Tan and Mitra (2010), the non-adaptive (NAD) weighting setting performed better than the adaptive ones (AAD and GAD) in our dataset. This might have happened because of the granularity of downloading cycles of our dataset.

While in the work of Tan and Mitra one download cycle was set to two weeks, in here the granularity of a download cycle is one day. Also, their dataset is composed of snapshots of the pages from the WebArchive[4] in period of 1 year, whereas we monitored pages by approximately 2 months. Lastly, the shortsighted-adaptive (SAD), which considers only the last visit, was the worse of the four weighting settings in our dataset. This shows once more that the history of the pages is an important source of information for prediction of future change behavior of pages.

Table 4.3 presents the average Change Ratio achieved by all methods. When looking to these results, we can see again that our GP4C approach produced score functions superior to all baselines. We observe that the results for *Rand* and *Age* are far away from the five baselines and the three GP4C approaches. Recall that in *Rand*, the scores are randomly chosen, whereas in *Age*, they are equal to the time elapsed since the page was last visited. These results stress the importance of good scheduling policies to improvement of resources usage.

Another important point to observe in Table 4.3, is that our GP4C method uses only the set of parameters adopted by the score function proposed by Cho and Garcia-Molina (2003b) (i.e., n, X), plus the time since last visit t. When comparing our results to the ones obtained by their proposed function, we can see that our method was able to produce fair better results, increasing the Change Ratio from about 0.64 to about 0.70, which represents an improvement of almost 10%. These results become more important when considering that our framework can easily derive other functions if more parameters are given as input.

The best baseline method in the experiments was NAD, with Change Ratio of about 0.69. This baseline uses other set of parameters that may provide more useful information about the updating behavior of the web pages. However, still our function was slightly better than it.

4.7 Performance

We measured the execution time of each phase of the GP4C process. Table 4.4 shows the total time spent in the whole GP4C process, as well as the time spent only in the population evaluation during training and validation phases. For each seed value, we report only the total time for all 5 folds in each phase, which means that the times include 5 training phases and 5 validations phases (one for each fold of the dataset).

We can observe that the time spent executing the crawl simulations corresponds

[4]http://archive.org

Table 4.3. Average Change Ratio for all days

Method	Fold 1	Fold 2	Fold 3	Fold 4	Fold 5	Average
Rand	0.185346	0.186671	0.185507	0.185537	0.185372	0.185687
Age	0.212147	0.213900	0.212727	0.213534	0.212928	0.213047
NAD	0.683776	0.693682	0.686584	0.694109	0.688084	0.689247
SAD	0.509400	0.518945	0.515104	0.523776	0.515605	0.516566
AAD	0.627277	0.642546	0.628031	0.642494	0.631454	0.634360
GAD	0.597471	0.607425	0.596756	0.605747	0.600741	0.601628
CG	0.639913	0.650251	0.637702	0.648119	0.643575	0.643912
$GP4C_{Best}$	0.703514	0.706889	0.711833	0.713094	0.693782	0.705822
$GP4C_{Sum}$	0.703375	0.705202	0.711441	0.707909	0.676050	0.700795
$GP4C_{Avg}$	0.703375	0.705202	0.711441	0.707909	0.688882	0.703362

to more than 99% of the total time of the GP4C process. The times for each seed value vary because different individuals are created across generations for different seed values, and the time to compute the scores depend on the structural complexity of each individual created. Although there are differences, the coefficient of variation of the execution times for all seeds is only 1.5%, which means that the seed values do not have a very large impact in the execution time.

Table 4.4. Execution time of each phase of the GP4C process in seconds.

	Seed values						
Phase	111	222	333	444	555	Total	%
Population Evaluation	43213	44321	44452	43095	44236	219316	99.624%
Validation	161	149	196	142	171	819	0.372%
Pop. Eval. + Validation	43374	44470	44647	43237	44407	220136	99.996%
Total time	43375	44472	44649	43239	44409	220144	100.000%

4.8 Summary of the Chapter

In this chapter we presented an experimental evaluation of GP4C using a dataset crawled from the Brazilian Web. We showed that the scheduling policies created using GP4C were able to create functions that outperform the baselines considered in most download cycles using only a very simple set of terminals. We also presented the execution time of the GP4C process to show that it is a viable solution to web crawlers.

Chapter 5

Conclusions and Future Work

In this work, we presented a Genetic Programming based framework to automatically generate score functions to be used by schedulers of web crawlers. We experiment our framework in the task of providing a score function to rank web pages according to their likelihood of being modified on the web since the last time they were crawled. The problem of finding such score function has been addressed by several authors in literature, and we compare the performance of the functions generated by our method to the best baselines we found.

The experiments have shown the ability of our framework to derive good competitive score functions for scheduling web pages updates. The functions generated were superior to all the baselines considered. For instance, the functions created using our $GP4C_{Best}$ method were statistically superior in most of the simulated download cycles.

An advantage of our framework is that it provides means for deriving new score functions if new features are provided as input, while previous work presented only punctual score functions, instead of a framework to generate them. For instance, our method would be able to incorporate information such as PageRank of the pages, cost for crawling, and even take the novelty of pages into account.

Further, the fitness function can also be adapted to take new crawling objectives into account, such as the importance of the pages, the likelihood of the page being presented to the final users in search results, the click through of pages in the system, chances of finding novel pages and so on. Once the input and the fitness functions are defined, our framework can be used to derive new functions. We intent to better explore such alternatives as future work.

Finally, in another research direction, we want to investigate the usage of our framework to derive score functions that allows the crawler to download important pages earlier in the crawl. The features in these case should be the ones usually

adopted in scheduling policies which pursue that objective, and the fitness function could be the weighted coverage metric previously presented.

We also plan to study the possibility of deriving functions to balance the two main objective functions of a scheduler: freshness and coverage. Besides that, there is a plenty of work regarding multi-objective optimization in Genetic Programming, which can also be explored to improve the scheduling policies generated.

Bibliography

Abiteboul, S., Preda, M., and Cobena, G. (2003). Adaptive on-line page importance computation. In *Proceedings of the 12th International World Wide Web Conference*, pages 280–290.

Alam, M., Ha, J., and Lee, S. (2012). Novel approaches to crawling important pages early. *Knowledge and Information Systems*, 33:707–734. ISSN 0219-1377.

Baeza-Yates, R., Castillo, C., Marin, M., and Rodriguez, A. (2005). Crawling a country: better strategies than breadth-first for web page ordering. In *Proceedings of the 14th International World Wide Web Conference*, pages 864–872.

Barbosa, L., Salgado, A. C., de Carvalho, F., Robin, J., and Freire, J. (2005). Looking at both the present and the past to efficiently update replicas of web content. In *7th ACM International Workshop on Web Information and Data Management*, pages 75–80.

Castillo, C., Marin, M., Rodriguez, A., and Baeza-Yates, R. (2004). Scheduling algorithms for web crawling. In *Proceedings of WebMedia and LA-Web*, pages 10–17.

Cho, J. and Garcia-Molina, H. (2000). Synchronizing a database to improve freshness. In *SIGMOD Record*, pages 117–128.

Cho, J. and Garcia-Molina, H. (2003a). Effective page refresh policies for web crawlers. *ACM Transactions on Database Systems*, 28(4):390–426.

Cho, J. and Garcia-Molina, H. (2003b). Estimating frequency of change. *ACM Transactions on Internet Technology*, 3:256–290. ISSN 1533-5399.

Cho, J., Garcia-Molina, H., and Page, L. (1998). Efficient crawling through url ordering. In *Proceedings of the 7th International World Wide Web Conference*, pages 161–172.

Cho, J. and Ntoulas, A. (2002). Effective change detection using sampling. In *28th International Conference on Very Large Data Bases*, pages 514–525.

Cho, J. and Schonfeld, U. (2007). Rankmass crawler: a crawler with high personalized pagerank coverage guarantee. In *Proceedings of the 33rd International Conference on Very Large Data Bases*, pages 375–386.

Coffman, E. G., Liu, Z., and Weber, R. R. (1998). Optimal robot scheduling for web search engines. *Journal of Scheduling*, 1(1).

da Costa Carvalho, A. L., Rossi, C., de Moura, E. S., da Silva, A. S., and Fernandes, D. (2012). Lepref: Learn to precompute evidence fusion for efficient query evaluation. *Journal of the American Society for Information Science and Technology*, 63(7):1383–1397. ISSN 1532-2882.

Dasgupta, A., Ghosh, A., Kumar, R., Olston, C., Pandey, S., and Tomkins, A. (2007). The discoverability of the web. In *Proceedings of the 16th International World Wide Web Conference*, pages 421–430, New York, NY, USA. ACM.

de Almeida, H. M., Gonçalves, M. A., Cristo, M., and Calado, P. (2007). A combined component approach for finding collection-adapted ranking functions based on genetic programming. In *30rd International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 399–406.

Douglis, F., Feldmann, A., Krishnamurthy, B., and Mogul, J. (1997). Rate of change and other metrics: a live study of the world wide web. In *USENIX Symposium on Internet Technologies and Systems*, pages 14–14.

Edwards, J., McCurley, K., and Tomlin, J. (2001). An adaptive model for optimizing performance of an incremental web crawler. In *Proceedings of the 10th International World Wide Web Conference*, pages 106–113.

Fetterly, D., Craswell, N., and Vinay, V. (2009). The impact of crawl policy on web search effectiveness. In *32nd International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 580–587.

Fetterly, D., Manasse, M., Najork, M., and Wiener, J. (2003). A large-scale study of the evolution of web pages. In *Proceedings of the 12th International World Wide Web Conference*, pages 669–678.

Guidolini, R. (2011). Detecção de réplicas de sítios web em máquinas de busca usando aprendizado de máquina. Master's thesis, Universidade Federal de Minas Gerais, Belo Horizonte.

Henrique, W. F. (2011). Verificação de unicidade de urls em coletores de páginas web. Master's thesis, Universidade Federal de Minas Gerais, Belo Horizonte.

Henrique, W. F., Ziviani, N., Cristo, M. A., de Moura, E. S., da Silva, A. S., and Carvalho, C. (2011). A new approach for verifying url uniqueness in web crawlers. In *Proceedings of the 18th International Conference on String Processing and Information Retrieval*, pages 237–248.

Järvelin, K. and Kekäläinen, J. (2002). Cumulated gain-based evaluation of ir techniques. *ACM Transactions on Information Systems*, 20:422–446.

Koza, J. R. (1992). *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press.

Lee, H.-T., Leonard, D., Wang, X., and Loguinov, D. (2009). Irlbot: scaling to 6 billion pages and beyond. *ACM Transactions on the Web*, 3(3):8.

Manku, G. S., Jain, A., and Sarma, A. D. (2007). Detecting near-duplicates for web crawling. In *Proceedings of the 16th International World Wide Web Conference*, pages 141–150.

Olston, C. and Najork, M. (2010). Web crawling. *Foundations and Trends in Information Retrieval*, 4(3):175–246.

Olston, C. and Pandey, S. (2008). Recrawl scheduling based on information longevity. In *Proceedings of the 17th International World Wide Web Conference*, pages 437–446.

Page, L., Brin, S., Motwani, R., and Winograd, T. (1998). The pagerank citation ranking: Bringing order to the web. In *Proceedings of the 7th International World Wide Web Conference*, pages 161–172.

Pandey, S. and Olston, C. (2005). User-centric web crawling. In *Proceedings of the 14th International World Wide Web Conference*, pages 401–411.

Poli, R., Langdon, W. B., and McPhee, N. F. (2008). *A field guide to genetic programming*. Lulu Enterprises Uk Limited.

Radinsky, K. and Bennett, P. (2013). Predicting content change on the web. In *6th ACM International Conference on Web Search and Data Mining*.

Tan, Q. and Mitra, P. (2010). Clustering-based incremental web crawling. *ACM Transactions on Information Systems*, 28:17:1–17:27. ISSN 1046-8188.

Tan, Q., Mitra, P., and Giles, C. L. (2007). Designing clustering-based web crawling policies for search engine crawlers. In *Proceedings of the 16th Conference on Information and Knowledge Management*, pages 535–544.

Wolf, J., Squillante, M., Yu, P., Sethuraman, J., and Ozsen, L. (2002). Optimal crawling strategies for web search engines. In *Proceedings of the 11th International World Wide Web Conference*, pages 136–147.