In-Rank mesh optimization for URL customized promotion in SEO

Stefan Duprey
Cdiscount
stefan.duprey@cdiscount.com

Fabien Jaunas
Cdiscount
fabien.jaunas@cdiscount.com

Abstract

Web site internal mesh optimization is at the very heart of search engine optimization. One prominent way to get the best web search engine visibility for your site is to build the adequate internal linking to promote your naturally popular pages. The definition of popular might be the transformation rate for e-commerce, the traffic from logs for a common site or even rather a specific semantic where you want to be visible. We here propose an algorithm to automatically compute the optimal internal mesh for your web site. The idea behind being simple : the higher you value your URL, the more in rank you want to give it. We tackle the challenges met both at a theory and software implementation level. We'll more specifically deal with big data issues for an e-commerce web site.

Keywords:

search engine optimization, e-commerce, page rank, in rank, mesh optimization, global optimization, thematic page rank, tf/idf, Levenshtein, Hadoop, Spark GraphX, Neo4j

I. Introduction

We here propose an original methodology to optimize the internal mesh of an e-commerce site. The idea behind is to promote the most successful URLs by increasing their in-rank. For the sake of simplicity, we define URLs successfulness in a first step as the URL traffic gathered from log. We'll see in a latter part how the same algorithm will extrapolate to any kind of metrics you cherish and we'll enter into e-commerce specifics. The frequency data is obtained from logs parsing tracking software and the in rank is computed using the famous page rank iterative algorithm. We want to find the optimal mesh, which maximizes for all URLs the matching between their traffic and their in-rank. The more successful a URL is, the more in-rank we want to give him through our optimal mesh.

We here detail the technical implementation of such an algorithm. The difficulty here is three-fold:

First the universe we deal with is discrete and vast. For a mesh with N nodes (a site with N URLs), the number of possible meshes (directed graph) is 2^{N^2}. We will also see that our objective function is non-linear and non-convex. Discrete non-linear optimization problems are among the most difficult optimization problems to solve. Not only optimum existence and uniqueness results are inexistent, but those problems appear generally to be computationnaly intensive to solve. For our case, exhaustive optimization would be far too computationally intensive. We have to find a proper heuristic based global optimization algorithm to cleverly browse through our universe. We here choose genetic algorithms among others following here [11]. But of course, as heuristic, this algorithm must not be trusted : it can get stuck into local minima and results will depend upon initial set-up and randomness stream seed. Parameters also have to be properly tweaked to proceed through our universe to avoid getting stuck into local minima. Other metaheuristics such as global search, multistart, particle swarm or simulated annealing as in [8] are also being tested. All seem promising. Robustness toward a general minimum, convergence fastness should here give the prevailing algorithm and its appropriate parameters.

Second the structure of a web site is usually well-defined and it is out of question to drastically change the already existing mesh. It is even truer for an e-commerce site : links are to be categorized regarding their incoming/targeting page type. Position within those pages is also relevant as most e-commerce sites dedicate specific zones for links to similar products, filtered lists and so on. Siloing navigation will also forbid links between distinct universe. So links can only be created within certain categories of pages and zones within those pages. In other terms our mesh will not be totally free and we'll have strong constraints for our mesh to comply with. That is good news : the less linking possibilities we got, the smaller is our optimization universe.

Third the actual size of an e-commerce site makes the implementation of a real-world industrial use case a technologically difficult problem. Either we implement it from scratch over a cluster dealing ourselves with concurrency and inter-processor communications, or we try not to reinvent the wheel and plug ourselves to existing big data cluster platforms. We'll here detail our technological choices.

II. Algorithm

Let $N \in \mathbb{N}$ be the number of nodes of our graph or rather the number of URLs of our site.

Let's

$$(X_i)_{i \in \{1,\dots,N\}}$$

denote the nodes of our directed graph.

Let's f be a function over our mesh, which gives for a URL our estimated potential value. This value is the very data we want to optimize our web site from. $(f(X_i))$ will also be named 'prominence vector' as it carries the specific metrics we want to optimize our mesh against.

$$\begin{array}{rcccc} f & : & (X_i)_{i\in\{1,\dots,N\}} & \to & \mathbb{R}^+ \\ & & x & \mapsto & f(x) \end{array} \quad (1)$$

Let's here say that our function is the traffic per URL. We'll see how we can improve this metrics in the next part.

Let's

$$(G_{ij}) \in \{0,1\}^{N\times N}$$

denote the vectorized adjacency matrix of our directed graph.

(G_{ij}) is either 1 or 0 if there is a link between URL i and j and our matrix is reshaped into a vector by concatenating each row.

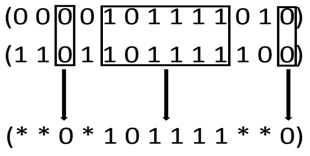
Genetic algorithm

Genetic algorithm is a search heuristic that mimics the process of natural selection as explained in [9]. A population of individuals to an optimization problem is evolved toward better solutions. In our case, each individual or chromosome $(C_i) \in \{0,1\}^{N\times N}$ here is represented as a bits array, which results from the very vectorization of our adjacency matrix (G_{ij}). An individual is a specific mesh.

At each generation, a candidate can either results from a crossover between two parents or from a self mutation.

For each new child to be produced, a pair of parents is selected for breeding from an eugenic pool from the previous generation. The newly created solution typically shares many of the characteristics of its parents. The child will keep the matching bits of the two parents and inherit randomly every other non-matching bits.

Child spawning from 2 parents crossover



For each new mutant to be produced, a single parent is selected from the same eugenic pool from the previous generation. The newly created mutant also shares many of the characteristics of its single parent. The mutation function is here standard : we just switch bits whose locations are randomly chosen and proportions equal a fixed defined mutation rate α :

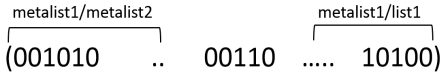
Mutation of an individual

(1 1 0 * 1 0 1 1 * 1 1 0 0)

New parents are selected for each new child and mutant, and the process continues until a new population of solutions of appropriate size is generated. These process ultimately results in the next generation population of chromosomes different from the initial generation. Generally the average fitness will have increased by this procedure for the population, since only the best organisms from the first generation are selected for breeding and mutating, along with a small proportion of less fit solutions. These less fit solutions ensure genetic diversity within the genetic pool of the parents and therefore ensure the genetic diversity of the subsequent generation of children. The number of newly bred fit individuals selected from the current population is a parameter to be fixed, The new generation of candidate solutions is then used in the next iteration of the algorithm. Commonly, the algorithm terminates when either a maximum number of generations has been produced, or a satisfactory fitness level has been reached for the population. The evolution usually starts from a population of randomly generated individuals and is an iterative process, with the population in each iteration called a generation.

Let's add here that taking into account our mesh constraints is here trivial : we just fix the value of blocs of links between forbidden categories from our vector. We just end up with a vector with a smaller number of freedom degrees : our vectorized adjacency matrix has large fixed null blocks resulting from locations where links can't be added.

Links categorization



We here present results for a small prototyping use case. We are provided with 39 URLs. We order and flag them as home, 2 meta lists, 4 lists and 32 products and we affect them the following prominence vector :

$$(X_i) = \{'home','metalist1','metalist2',$$

$$'list1','list2','list3','list4',$$

$$'product1','product2',...,'product32'\};$$

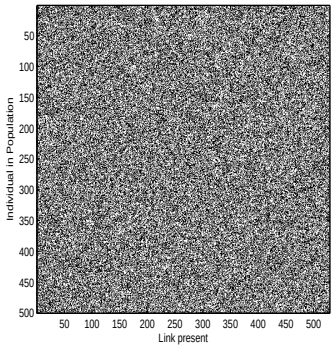
$$f(X_i) = [100; 80; 55;$$

$$40; 35; 25; 20;$$

$$4; 5; ...; 3]$$

Notice that the prominence metrics vector we chose mirrors the labels we gave. We hope to find back the natural arborescent structure from our algorithm by giving appropriate values. The higher in the arborescence, the greater the value in the prominence vector. We end up with a bunch of small value for the product leaves. Let's now generate our initial population through sheer randomness. Let's visualize this population with this picture where x axe represents an individual chromosome, while y axe runs over all the individuals of our population. We here don't have constraints on our links, so you won't see large null streaks resulting from fixed null blocks in our adjacency matrix :

Initial population



For each generation, the fitness of every individual in the population is evaluated. Our fitness function is defined as follows : First we compute the standard page rank of the individual matching mesh following [6] : The first stage is the page rank initialisation for all our nodes:

$$\forall u \; PR(u) = \frac{1}{N} \tag{2}$$

We then iterate to find the fixed point probability P solution of the well know equation where c is the damping factor set around 0.85 :

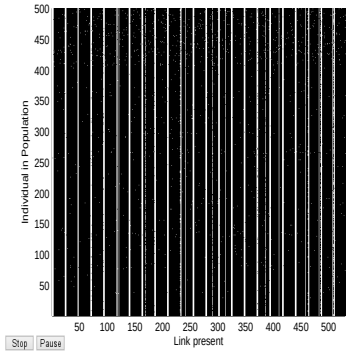
$$PR(u) = \frac{(1-c)}{N} + c \times \sum_{v \to u} \frac{PR(v)}{card(\{v \to u\})} \tag{3}$$

The fitness is usually the value of the objective function in the optimization problem being solved. The fitness function for a mesh (X_i) is defined as the sum over all nodes of the product of the local prominence metrics and the computed in rank : $\sum_{i=1}^{N} f(X_i) \times PR(X_i)$, the idea being that by weighing the local prominence metrics with the in rank we'll find the best mesh matching in-rank and the local prominence metrics and this for all nodes as the objective is the whole sum. So we end up with this generic optimization problem.

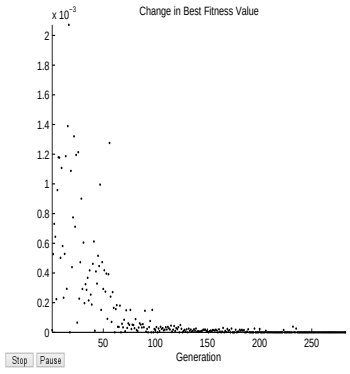
$$\max_{(G_{ij}) \in \{0,1\}^{N \times N}} \left\{ \sum_{i=1}^{N} f(X_i) \times PR(X_i) \right\} \tag{4}$$

This is this very same global optimization problem, which can be formulated using different algorithms : simulated annealing, global search, multistart. Let's here detail the results for our small case toy example of 39 URLs. For a population of 500 individuals, a mutation rate of 0.2% and a proportion of less fit solutions of 2%. We define our convergence threshold as the fitness function evolution tolerance : the algorithm runs until the average relative change in the fitness function value over stall generations is less than the specified threshold 10^{-6}. Our algorithm converges after roughly 250 iterations for our toy site example. Note here that the reached minima seems global, as multiple starts would lead to the same structure. Let's now visualize our population at convergence and the evolution of the fitness function for our best element at each generation.

Population at convergence

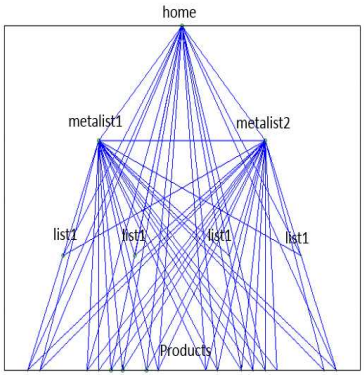


Best chromosome fitness function evolution



The structure found is as we expected arborescent : home linking to metalists, metalists linking to lists and lists to products.

Optimal site structure



III. An e-commerce approach

For an e-commerce web site, traffic might not always coincide with the URLs you deem as more profitable to you. On the very contrary, you may even actually rather want to rebalance some traffic discrepancies. Of course, the methodology presented above is still valid. You here just have to change the prominence vector you want to optimize. In that case, you would for instance rather optimize $\left(\frac{1}{traffic(i)}\right)_{i\in\{1,\dots,N\}}$. But to get a real sense of potential return on investment, we must here switch to a new paradigm. Each page will be matched with a specific keywords expression to rank upon. And the potential return on investment for the keywords expression will be assessed with both search engine data (click through rate, search volume) and e-commerce tracking data (transformation rate). To push potentially high return on investment pages, we use the following objective function :

$$\sum_{i\in Keywords} SV\,(i)\times CTR\,(position\,(i))\times CR\,(i)\times P\,(i) \tag{5}$$

where $position\,(i)$ is the estimated position in search engine results coming from the modification of our new mesh,

$SV\,(i)$ is the search volume for the keywords i estimated by the search engine.

and $CTR\,(i)$ is the click through rate for an URL at the position place in the search engine results.

The difficult part here is how to estimate the new position in search engine results for our new mesh. This is here hardly conceivable to imagine a feed-back loop with a real search engine indexation. This would drastically slow our algorithm. The best approach here is to build and etalonnate a machine learning algorithm to model the relationship between local in-rank $PR\,(i)$ and $position(i)$ as in [7].

IV. Semantic in-rank

One last digression here is about a semantic enhancement to our algorithm. We have seen that we can force siloing navigation by enforcing constraints on our mesh. A natural way to enlarge this constraints not only to distinct universes but also between products within the same universe would be to use a semantic similarity metrics. As most e-commerce web sites tend to overlink web pages between themselves, this would allow us to enlighten and rebalance our links according to their semantic reliability. We have here two ways to implement this.

Either by incorporating our semantic coefficient directly in the computation of the page rank algorithm that computes $PR(X_i)\forall i$.

Or by enforcing the absence of links between two nodes when the semantic similarity between them is below a certain fixed similarity threshold t.

$$\max_{(G_{ij})\in\{0,1\}^{N\times N}\ G_{ij}=0\ if\ CS(ij)\leq t} \left\{\sum_{i=1}^{N} f\,(X_i)\times PR(X_i)\right\} \tag{6}$$

where $CS(ij)$ is a semantic distance between the two linked pages i and j. $CS(ij)$ can be defined very easily as the scalar product of the tf/idf vectors of the product descriptions whose weight are defined by the well known formula :

$$w_{ik} = \frac{tf_{ik}\log\left(\frac{N}{n_k}\right)}{\sqrt{\sum_{k=1}^{t}(tf_{ik})^2\log\left(\frac{N}{n_k}\right)^2}} \tag{7}$$

where t is the number of terms in the production descripton, tf_{ik} is the frequency of term k in the product i description, n_k is the frequency of term k in the corpus of all product descriptions and N the total number of products. In an e-commerce world this formula might miss similarity between very specific product description as they will appear as distinct terms in the tf/idf vector (for instance the battery DCI_6020 vs DCI_7020). That can be overcome by adding a Levenshtein distance to terms with the highest inverse document corpus frequency between the two vectors.

V. Big data implementation

We'll detail in this section the technological implementation of a real world use case. To be scalable is a must for any e-commerce site which intends to grow ! We use open source tools Apache Hadoop [1] and Apache Spark [4] to solve these big data problems in a scalable way.

Solutions and performances

First, let's remark here that an easy and very efficient way to improve our algorithm performance

would be to improve the in rank computing function. Its performance is key for an efficient algorithm, as it is computed recurrently for each generation over all the population. The idea here would be to compute an approximated but faster in rank function as in [16].

Second, the double iterative nature of our algorithm (page rank computations and global optimization are both naturally iterative) makes Hadoop Map/Reduce paradigm in [12] too slow for our purpose. In contrast to Hadoop's two-stage disk based Map/Reduce paradigm, Spark's in-memory primitives provide the needed performance. By allowing user programs to load data into a cluster's memory and query it repeatedly, Spark is well suited for our purpose ([14] and [13]).

The problem here is that the very same restrictions that enable graph-parallel systems as GraphX to achieve substantial performance gains also limit their ability to express many of the important stages in the feeding process of new graph to be computed. As Spark GraphX is optimized for iterative diffusion algorithms like PageRank it is not well suited to more basic tasks like constructing the graph, modifying its structure, or expressing computation that spans multiple graphs.

Technical implementation

Our internal mesh is stored in Neo4j [2]. It allows us to solve analytical problems that relational databases struggle to solve in a flexible way (mainly recursive mesh exploration with CYPHER). We then use a specific in-house toolkit to create a job to export subgraphs from Neo4j to Apache Hadoop HDFS. This allows us to pilote the distributed algorithm from a small client (the creation of the graph population and its evolution). Note here that the bits vector representation of our mesh is here indeed changed to a sparse adjacency matrix for the page rank computation. It will then start a distributed graph processing algorithm using Scala [3] and Spark's GraphX module [5]. The GraphX algorithm is serialized and dispatched to Apache Spark for processing. Once Apache Spark job is done, the results are written back to HDFS as a Key-Value list of property updates to be applied back to Neo4j. Neo4j is then notified that a property update list is available from Apache Spark on HDFS. Neo4j batch imports the results and applies the updates back to the original graph, where the fitness function can now be computed for the individual.

VI. Conclusion and future works

Mesh optimization is a key concept for SEO. Numerical experiments on a small case example show the relevance and the possibility to actually improve your site mesh to push forward pages accordingly to a prominence metrics which is yours to define. The difficulty here does not lie in the theorical conception of the algorithm, but rather in its distributed implementation on a big data platform. The choice of open source solutions is here highlighted. It allows a fast ramp-up for anyone to a high level technological state.

References

[1] *Hadoop*. http://hadoop.apache.org.

[2] *Neo4j*. http://neo4j.com.

[3] *Scala*. http://www.scala-lang.org.

[4] *Spark*. https://spark.apache.org.

[5] *Spark GraphX*. https://spark.apache.org/graphx/.

[6] S. Brin and L. Page. The anatomy of a large-scale hypertextual web search engine. *WWW*, 1998.

[7] Qin T. Liu T.Y. Tsai M.F. Li H. Cao, Z. Learning to rank: from pairwise approach to listwise approach. *Proc. of the 24th Intl. Conference on Machine Learning*, page 129136, 2007.

[8] C. Sanchez F. Xhafa, A. Barolli and L. Barolli. A simulated annealing algorithm for router nodes placement problem in wireless mesh networks. *Simulation Modelling Practice and Theory*, 19:pp. 22762284, 2011.

[9] D. E. Goldberg and J. H. Holland. *Genetic algorithms and machine learning. Machine Learning.* Springer, 1988.

[10] D. E. Goldberg and J. H. Holland. *Big Data Benchmarks, Performance Optimization, and Emerging Hardware.* Springer, 2014.

[11] Keivan Kianmehr Reda Alhajj Jon Rokne Iltae Lee, Negar Koochakzadeh. *Integrating Genetic Algorithms and Fuzzy Logic for Web Structure Optimization*, volume 12. Springer, 2010.

[12] Sanjay Ghemawat Jeffrey Dean. Mapreduce: a flexible data processing tool. *Commun. ACM*, 53:72–77, 2010.

[13] Tathagata Das Ankur Dave Justin Ma Murphy McCauley Michael J. Franklin Scott Shenker Ion Stoica Matei Zaharia, Mosharaf Chowdhury. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. *Journal of University of California, Berkeley*, 2012.

[14] R. Rodrigues U. A. Acar P. Bhatotia, A. Wieder and R. Pasquin. Mapreduce for incremental computations. *ACM SOCC*, 2011.

[15] Vapnik V.N. *Statistical Learning Theory*. Wiley, 1998.

[16] Li-Tal Mashiach Ziv Bar-Yossef. Local approximation of pagerank and reverse pagerank. *Proceedings of the 17th ACM Conference on Information and Knowledge Management (CIKM)*, pages PP. 279–288, 2008.