In-Rank mesh optimization for URL customized promotion in SEO

Stefan Duprey
Cdiscount
stefan.duprey@cdiscount.com

Second Author
Second University
second.author@university2.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 our web site. The idea behind being simple : the higher you value your URL (potential ROI), 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, e-commerce, page rank, in rank, mesh optimization, global optimization

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 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 usually reveal 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 tweak through our universe. We here choose genetic algorithms among others. Simulated annealing, particle swarm, global search have been tested and also works on the small case to be presented. But of course, for a real world e-commerce example, those algorithms must not be trusted : they might get stuck into local minima. And as we use parameters and randomness for our algorithm, the returned solution might change from one try to another. Parameters have to be properly tweaked to proceed through our universe avoiding getting stuck into local minima.

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, list of products and so on. So links can only be created within certain categories of pages and zones within our page. 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, our algorithm dealing ourselves with concurrency and inter-processor communications, or we try not to reinvent the wheel and plug ourselves to existing big data platform. 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.

$$\begin{array}{rccc} f & : & (X_i)_{i \in \{1,\dots,N\}} & \rightarrow & \mathbb{R} \\ & & x & \mapsto & f(x) \end{array} \tag{1}$$

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

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

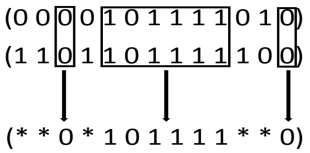
denote the adjacency matrix of our directed graph.

(G_{ij}) is either 1 or 0 if there is a link between URL i and j.

Genetic algorithm methodology

Genetic algorithm is a search heuristic that mimics the process of natural selection. 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 represented as a bits array, which results from the vectorization of the adjacency matrix (G_{ij}). At each generation, a candidate can either result from a crossover between two parents or from a self mutation. The crossover function which breeds a child from two parents is defined as follows : The child will keep the matching bits of the two parents and inherit randomly every other non-matching bits. For every two parents, we spawn two children :

Child spawn from 2 parents crossover



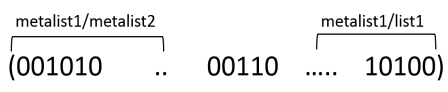
The mutation function is even simpler : we just switch bits whose locations are randomly chosen and proportions equal a fixed mutation rate :

Mutation of an individual

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

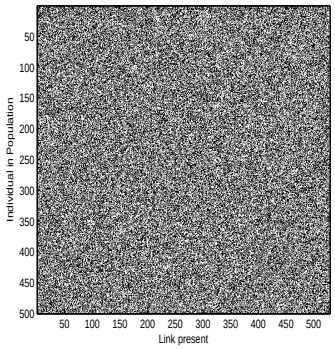
The evolution usually starts from a population of randomly generated individuals and is an iterative process, with the population in each iteration called a generation. Let's add here that taking account our mesh constraints is here trivial : we just fix the value of blocs of links between forbidden categories from our vector. We just end up with a vector with a smaller number of freedom degrees : our adjacency matrix is square but with large fixed null blocks resulting only from locations where links can be added.

Links categorization



Our initial population is then indeed sheer randomness :

Initial population



For each individual, we evaluate a fitness function. Our fitness function is defined as follows : First we compute the standard page rank of the individual matching mesh, which is defined as follows : The first stage is the page rank initialisation for all our nodes:

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

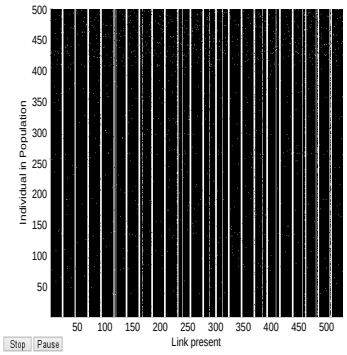
We then iterate to find the fixed point solutions of the well know equation :

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

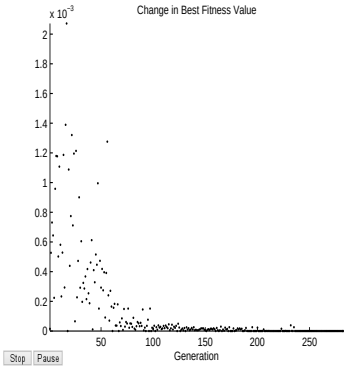
In each generation, the fitness of every individual in the population is evaluated; the fitness is usually the value of the objective function in the optimization problem being solved. The fitness function for a mesh (X_i) is defined as the sum over all nodes of the product of the local importance metrics and the computed in rank : $\sum_{i=1}^{N} f(X_i) \times PR(X_i)$, the idea being that by weighing the local importance metrics with the in rank we'll find the best mesh matching in-rank and the local importance metrics and this for all nodes as the objective is the whole sum. The more fit individuals are stochastically selected from the current population, and each individual's genome is modified (recombined and possibly randomly mutated) to form a new generation. 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.

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

Population at convergence



Best chromosome fitness function evolution



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. You indeed on the very contrary want to rebalance some traffic discrepancies to push high ROI potential URLs. Of course, the methodology presented above is still valid. You here just have to change the vector you want to optimize.

Page rank thematic little topo here.

IV. Big data implementaion

The double iterative nature of our algorithm (page rank computations and optimization are both naturally iterative) makes Hadoop Map/Reduce paradigm 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. We'll detail the technological implementation of a real-world use case.

The in rank fitness function is computed recurrently over the whole population. To improve its performance is key for an efficient algorithm. The idea here would be to computed an approximated in rank using [].

The sparsitiy of the matrix has to be addressed.