GraphOpt: a Framework for Automatic Parameters Tuning of Graph Processing Frameworks

1st Muaz Twaty
Research and Development Department
EURA NOVA
Mont-Saint-Guibert, Belgium
muaz.twaty@euranova.eu

2nd Amine Ghrab
Research and Development Department
EURA NOVA
Mont-Saint-Guibert, Belgium
amine.ghrab@euranova.eu

3rd Sabri Skhiri
Research and Development Department
EURA NOVA
Mont-Saint-Guibert, Belgium
sabri.skhiri@euranova.eu

***Abstract*—Finding the optimal configuration of a black-box system is a difficult problem that requires a lot of time and human labor. Big data processing frameworks are among the increasingly popular systems whose tuning is a complex and time consuming. The challenge of automatically finding the optimal parameters of big data frameworks attracted a lot of research in recent years. Some of the studies focused on optimizing specific frameworks such as distributed stream processing [1] [2], or finding the best cloud configurations [3], while others proposed general services for optimizing any black-box system [4]. In this paper, we introduce a new use case in the domain of automatic parameter tuning: optimizing the parameters of distributed graph processing frameworks. This task is notably difficult given the particular challenges of distributed graph processing that include the graph partitioning and the iterative nature of graph algorithms. To address this challenge, we designed and implemented *GraphOpt*: an efficient and scalable black-box optimization framework that automatically tunes distributed graph processing frameworks. GraphOpt implements state-of-the-art optimization algorithms and introduces a new hill-climbing-based search algorithm. These algorithms are used to optimize the performance of two major graph processing frameworks: Giraph and GraphX. Extensive experiments were run on GraphOpt using multiple graph benchmarks to evaluate its performance and show that it provides up to 47.8% improvement compared to random search and an average improvement of up to 5.7%.**

***Index Terms*—Distributed Graph Processing; Parameters Tuning; Black-box Optimization**

I. Introduction

Graphs are powerful mathematical structures used to model a large variety of domains and problems. Optimizing graph operations is a necessary task, especially with the rise of large and complex graphs in popular applications such as social or biological networks. To efficiently handle these large graphs, a plethora of graph-processing frameworks were proposed such as Giraph and GraphX [5]–[9]. However, these frameworks have each a large set of specific parameters that require expert knowledge to fine-tune them before efficiently performing large graph jobs. Some frameworks have more than 180 parameters to set, as in GraphX, which makes the complete search, in the exponentially growing search space, infeasible. Therefore, the goal of this paper is to use optimization algorithms in order to automatically and efficiently find the best configurations of a given graph processing framework.

For this paper, we chose to focus on two popular and actively maintained distributed graph processing frameworks: Giraph and GraphX. These distributed frameworks are designed to handle large distributed graphs that do not fit in a single machine. They provide libraries to perform basic operations and computations on graphs and allow developers to define their implementations for new graph algorithms and custom functions. The main advantage of using such frameworks is that it abstracts all the distribution of the algorithms and hide complex tasks such as graph partitioning and machine failover. For example, Giraph is a vertex-centric computing model that iteratively executes a user-defined program over the vertices of the graph. The user-defined vertex function typically includes data from adjacent vertices or incoming edges as an input, and the resultant output is communicated along outgoing edges. This makes the implementation of a distributed graph algorithm much easier. However, these distributed framework have a large number of parameters specific to the distribution which makes their tuning more difficult. Parameters such as the number of workers, memory per worker and other communication parameters have a big effect on performance (as shown in section V-A). Not tuning these parameters or leaving them to their default values generally leads to non-optimal performance. As shown in the state of the art [3], some of the existing works aim to optimize the hardware architecture of a distributed system. However, this is out of the scope of our work. We assume that the hardware resources (the computation cluster resources) are fixed and the goal is to optimize the parameters of the framework rather than the hardware architecture.

To tackle this problem, we model the performance of a distributed framework as a black-box function. This performance function with respect to the parameters of the framework has no analytical expression and we cannot compute a closed form solution to optimize it or to use gradient-based techniques because we do not have access to its derivatives. The contributions of our work can be summarized as follows:

- We design and implement *GraphOpt*: an efficient and scalable optimization framework to automatically tune

The elaboration of this scientific paper was supported by the Ministry of Economy, Industry, Research, Innovation, IT, Employment and Education of the Region of Wallonia (Belgium), through the funding of the industrial research project Jericho (convention no. 7717).

graph-processing frameworks. At this point, the framework supports two major graph processing frameworks and three optimization algorithms.
- We introduce a new implementation of a hill-climbing based search algorithm. This implementation includes a new method for generating neighbors and does not use a prediction model.
- We make extensive experiments to evaluate the performance of each optimization algorithm on different graph jobs to demonstrate the efficiency of *GraphOpt*.

This paper is structured as follows: after this introduction, in Section II, we explain in detail the preliminary topics related to our work. After these preliminaries, in Section III, we review the state of the art in the domain of parameter tuning. In Section IV, we present our contribution and explain the choices we made when building *GraphOpt*. Extensive experiments with a discussion are presented in Section V. These experiments show a promising improvement compared to random search when optimizing some of the benchmarks we used. We conclude our paper in Section VI.

II. Preliminaries

In this section, we will review some basic definitions and concepts related to our work. These concepts include details about the chosen frameworks and some basic definitions and concepts that we will refer to in the paper.

A. Giraph

Giraph is an open source implementation of $Pregel$ [12], which is Google's implementation of a graph processing architecture inspired by bulk synchronous parallel model (BSP). Giraph is one of the most famous and widely used graph processing frameworks. Giraph implements a synchronous timing and a message passing communication protocols. The execution model of Giraph is the gather-apply-scatter model. No partitioning algorithm is implemented in this framework. This means that partitioning of the graph is based on the partitioning of the graph file in the Hadoop distributed file system (HDFS).

We chose the parameters to be optimized based on our knowledge of the framework, and the feedback from the Giraph developers community (as shown in section V-A). Table I contains the list of Giraph's parameters that we consider for optimization.

B. GraphX

GraphX is a graph processing framework implemented as an extension to Spark [5]. Resilient Distributed Datasets (RDDs) are collections of elements that Spark can process in parallel. To process graphs in Spark, GraphX creates a vertices RDD and an edges RDD. Similarly to Giraph, GraphX uses a synchronous timing protocol and a message passing communication protocol. GraphX implements multiple partitioning schemes and allows the users to define their own partition schema. The list of parameters to tune was chosen based on expert knowledge. Table II contains the list of Spark parameters which we will optimize.

TABLE I
The list of Giraph parameters which we are optimizing

Parameter	Default value
Number of workers (-w)	-
Mapreduce Map Memory	1024Mb
Client Receive Buffer Size	32Mb
Client Send Buffer Size	512Mb
Async Message Store Threads	1
Channels Per Server	1
Netty Client Execution Threads	8
Netty Client Threads	4
Netty Server Execution Threads	8
Netty Server Threads	16
Java Garbage Collector	-

TABLE II
The list of GraphX parameters which we are optimizing

Parameter	Default value
Spark Executor Memory	1024Mb
Spark Shuffle Compress	False
Spark Shuffle File Buffer	32Mb
Spark Speculation	False
Spark Default Parallelism	12
Java Garbage Collector	-

C. Search Spaces

The search space of a given problem represents the set of all possible solutions. In our case, the search space represents all the possible combinations of the values of all parameters. Clearly, this search space is exponentially growing when increasing the number of parameters. Search spaces can be finite or infinite. For example, if one of the parameters takes a real value, the number of all possible solutions is infinite. One of the methods used to overcome this problem is *value discretization*. This method is widely used when dealing with infinite search spaces or even with huge finite search spaces. The main idea is to bound the range of the value for each parameter and to fix the number of possible values that it can take. Most of the time, these values are spread evenly over the bounded range. In the case of finite search spaces, this is useful to reduce the size of the search space.

D. black-box Functions

A black-box function is a function for which we can only observe the input and output. We do not have an analytical expression that expresses the function and we do not have access to its derivatives. This modeling approach is useful when dealing with very complex systems. For example, a function representing the performance of a neural network with respect to its architecture parameters. Modeling the performance of distributed frameworks as a black-box function is widely used in the state-of-the-art [4] [2].

E. Latin Hypercube Sampling

This sampling technique is a statistical method that generates near-random and multidimensional points in a given space. This requires to specify in advance the number of required points to generate. In the case of discrete variables, it ensures that no two points share the same value of any of the

dimensions. This generates a representative sample from the space of the dimensions. Unfortunately, as explained in [1], Latin Hypercube Sampling (LHS) is not enough to guarantee a good spread in the search space. Maximin Latin Hypercube Sampling [13] uses LHS to generate better samples. The idea is to generate multiple sets of samples $S_1, S_2, .., S_w$, where each set of samples is a set of points generated using LHS. We chose the best set such that:

$$S^* = \arg\max_{1 \leq i \leq w} \min_{X^j, X^k \in S_i, j \neq k} dis(X^j, X^k)$$

Where dis is a distance function. In simpler words, this equation chooses the set of samples where the distance between samples is hopefully large enough to ensure a wide coverage.

A common modification of the LHS is to generate the samples in continuous settings. For example, the value of each dimension is generated in the range $[0, 1]$. Then, we map these values to the discrete representation. The reason for this modification is that the original version of LHS has a limit on the number of points that it can sample at once. For example, say that the value of one of the dimensions d_i can take one out of three possible values only. This means that LHS cannot sample more than three samples. Otherwise, two points must share the value of the dimension d_i.

III. Related Work

Automatic parameter tuning has been studied extensively in recent years. [1] focused on tuning the parameters of a streaming framework, the special nature of a streaming job -the execution time is unbounded: the input is unbounded- required a special study in which the authors focused on the *throughput* and the *latency* of the system. Other studies, like [3], focused on tuning the hardware architecture of a cloud distributed computation cluster. They worked on choosing the "best" type of hard-desks, processors, and RAM. Also, hyper-parameters of a machine learning algorithm was the main topic of some other studies [4].

We can divide these studies into two groups of approaches. The first approach depends on a prediction model to guide the search. This model will guess the execution time of a given job under a specific choice of parameters values. Some studies build a prediction model during the search process [1]. Other studies try to build a general prediction model before the search process [14]. All the training is done before the search starts. Creating a general model includes features extraction from each type of job and from the input data files. The second approach does not use a prediction model. The study of black-box function optimization is a sub-domain of this kind of approaches. Here we do not perform any kind of job feature extraction. We assume that we can only compute the execution time of the target function given its parameters values. This means that this kind of technique can optimize any function without any domain knowledge about the function or its properties.

Many optimization techniques have been used in the state-of-the-art of systems optimization. Bayesian Optimization [15], Hill-climbing and Covariance Matrix Adaptation Evolution Strategy are examples of these techniques. Some techniques are suitable when using a prediction model (like Hill-climbing-based search algorithms). Other techniques are more suitable under the sub-domain of black-box function optimization (like Bayesian Optimization).

IV. System Description

As the performance of a graph processing framework has a complex relation with the framework parameters, we chose to represent it as a black-box function II-D. The performance function, with respect to the parameter of the frameworks, does not have an analytical expression. Thus, we cannot compute the derivatives with respect to the parameters. We assume that the function is smooth: a small change in the value of the parameters should not result in a huge change in the performance. We implemented three optimization algorithms to apply to the target function (the performance of the processing framework). To compute the performance of the framework under specific values of the parameters, we run the actual graph job and we compute the execution time. The input data file -the input graph- is fixed along the whole optimization process. The same for the graph algorithm (the graph job). Extracting features for the input data files and for the job is not in the scope of *GraphOpt*. We assume that both are fixed. Tuning a specific graph job for specific input data files is useful when tuning frequently executed graph jobs on huge data files. For example, optimizing a recommendation system for a video-on-demand service. This kind of system should run the algorithm frequently (every 15 minutes for example) while keeping the run time of the algorithm under a given threshold (10 for example). In this scenario, it is relevant to use such approaches. The input data is not changing radically between different runs and we are applying the same algorithm every time.

Working in a distributed environment results in noisy measurements: running the same task under the same values of the parameters could result in different execution times. To produce more accurate measurements, we chose to run each task multiple times and consider the average execution time. The same method was used in [2]. More details are explained in section V-A.

A. Designing the Search Space

There exist two types of parameters (of Giraph and GraphX). The first type is categorical. The second type is discrete: there exists a range of possible integer values that the parameter can take. For the second type of parameter, if we want to consider all the possible values of each parameter, the search space will be unnecessary large. Plus, the effect of changing one of these parameters by one step will result in a negligible effect. For example, changing the Mapreduce Map Memory parameter from 300 MB to 310 MB will probably result in no effect at all. A design choice that has been

considered in all the related work -that we know of- is to limit the values which these parameters can take. For example, the Mapreduce Map Memory parameter can take one of the values in the list: $\{512\,\text{MB}, 1024\,\text{MB}, 1536\,\text{MB}, 2048\,\text{MB}\}$. This will decrease the size of the search space.

One problem we encountered is that some of the configurations cause the job to fail. For example, setting inefficient memory for the workers could cause an Out Of Memory (OOM) run time error and cause the failure of the job. In this case, we chose to return a very large value for the performance of this configuration. Our intuition is that this will guide the search away from these "bad" configurations. Another problem is that some of the configurations result in a very bad performance. Sometimes, this performance is 10 times slower than the performance under the default values of the parameters. This makes the search overhead larger. To prevent this problem, we chose to set a time limit on each job run. When exceeding this limit, the job is killed and returns a large value as well.

To reduce the number of failing jobs (and jobs that exceed the time limit), we experimented to detect the configurations which cause this failure. The idea of this experiment is similar to Coordinate Descent. However, the goal is not to find the best configurations but to find configurations that cause the failure. For each parameter, we save the value for which the job fails. For example, for the Mapreduce Map Memory parameter, this experiment will find the minimum memory required for the workers. All values which are less than this are eliminated from the parameters range. This experiment reduced the number of failing jobs in the optimization phase. It also reduced the size of the search space. However, there are still some failing jobs due to the second order effect between parameters. More details about this second order effect between parameters are explained in the experiments section V-A.

B. Random Search

Since there exists no similar work related to graph-processing frameworks, that we know of, we compare our results to random search results. In this case, Random Search is a common baseline used to compare to other algorithms. Usually, the idea is to give the same budget -in terms of the number of iterations- for random search and other algorithms and comparing the results in terms of accuracy.

C. Coordinate Descent

This search technique assumes that the parameters are independent -they do not have any second-order effect or higher- which is a very strong assumption. This technique tunes each parameter separately: when tuning the first parameter, it tries all the possible values of this parameter while fixing all the values of the other parameters to their default values. Once we are finished with the first parameter, we fix it value to the best one found and we move to the second parameter. We do the same for all parameters. This technique is used when the search space is very large. Using this technique could lead to suboptimal solutions. Clearly, this approach does not explore the whole search space. This approach was used in [16] to tune Map Reduce jobs and in [17] to tune web services.

D. Hill-Climbing Algorithms

Hill-climbing-based algorithms are greedy algorithms that assume that good points in the search space are close to each other. It also assumes that the function is smooth. [1] implemented a version of the Hill-climbing algorithm to search for a good configuration of a Stream Processing Framework. This implementation was based on the implementation of [17]. The main component of this implementation is a prediction model: a polynomial fitting of the points sample so far. This model was used to predict the potential best configuration to guide the search.

In the case of a continuous search space of a function for which we have access to its derivatives, the algorithm starts from a random point. Then, it computes the derivatives with respect to each parameter. The value of the derivatives will allow the algorithm to choose the next point to evaluate. However, if we do not have access to the function derivatives, we can try multiple points in the neighborhood of the current point and choose the best one as the next point. The neighborhood of a point in the search space usually means points that are close to the current point. Two examples of closeness measures are the Euclidean Distance and the Gaussian kernel.

Using a prediction model to guide the search in this kind of algorithm is suitable [1]. However, this is out of the scope of our work. Since we are dealing with expensive black-box functions, we do not try to extract any features for the graph jobs. This is the scope of another track in the Research Department of the company. Dealing with expensive functions is another reason that prevents us from using a prediction model. We can not obtain enough data to build a good model. We are targeting a small number of evaluations (between 20 and 40). Our implementation of the Hill-climbing search algorithm assumes that "good" points in the search space are close to each other. Algorithm 1 explains our choices in detail.

The initialization method used is Maximin Latin Hypercube Sampling II-E. As for the method of generating neighbors, our method depends on Maximin Latin Hypercube Sampling and on a distance function for which we used Euclidean Distance.

E. Bayesian Optimization

Bayesian Optimization is an active learning technique which aims to perform a global optimization on a given function. The goal is to optimize the sequence of the points:

$$x_m \in \mathbb{R}^n, m = 1, 2, ..$$

Such that it converges to x^*, the global optimum of the function. At each step $i + 1$, the choice of the point x_{i+1} is based on the data observed previously:

$$D_i = \{(x_1, y_1), (x_2, y_2), .., (x_i, y_i)\}$$

The mathematical representation of the problem is defined as a minimization/maximization of an unknown function with n parameters:

Algorithm 1 A pseudo code for our implementation of a hill-climbing-based search algorithm.

```
getInitializingPoints(m)

while Evaluation Budget is not finished do
    curBestCon = GetBest()
    neighborhood = GenerateNeighbors(curBestCon,k)
    if IsVisited(neighborhood) then
        continue                          ▷ Skip loop iteration
    Evaluate(neighborhood)
end while

procedure GETBEST
    return the best configuration -found so far- which has
           not been fully expanded yet

procedure GENERATENEIGHBORS(C,k)
    sSet = GenerateSparseSet()
    return the k closest configurations to C from sSet

procedure GENERATESPARSESET
    for i in 1,2,3 .. 10 do
        generate a set S_i of 50 configuration using LHS
    sparseSet = arg max_{S_i} MinDistance(S_i)

procedure MINDISTANCE(S)
    return the minimum Euclidean distance between any
           pair of configurations.
```

$$x^* = \arg\min_{x \in \mathbb{R}^n} f(x)$$

Since we do not have an analytical expression of the function, we assume that the function f belongs to a family of functions $f \in F$ and that it is sampled from a probability distribution $P(f)$. At step i, the optimization can be defined as minimizing the expected value of the function:

$$x_{i+1} = \arg\min_{x \in \mathbb{R}^n} \mathbb{E}_{P(f)}[f(x_{i+1})] = \arg\min_{x \in \mathbb{R}^n} \int_F f(x_{i+1}) dP(f)$$

This can be improved as data arrives. When we want to choose the next point to sample X_{i+1}, we can update the prior distribution with the observed data D_i. The update of the prior distribution is given by Bayes rule:

$$P(f|D_i) = \frac{P(D_i|f)P(f)}{P(D_i)}$$

This results in the following equation:

$$x_{i+1} = \arg\min_{x \in \mathbb{R}^n} \int_F f(x_{i+1}) dP(f|D_i)$$

As explained in [15], Bayesian Optimization uses a surrogate model (a Gaussian Process for example) to represent the behavior of the target function. We start with a prior distribution $P(f)$ which is updated as data arrives. This results in the Bayesian posterior $P(f|D_i)$, which represents our -updated- understanding of the target function.

Acquisition Functions are heuristics used to choose the point -in the search space- which will be sampled next. These functions keep a trade-off between exploration and exploitation. To do pure exploration means the tendency to select a part of search space which has not been sampled frequently. On the other hand, doing pure exploitation means the tendency to select a part of the search space where we already sampled the best points so far. There exist multiple famous and widely used acquisition functions. Some of those are:

1) Probability of Improvement (PI),
2) Expected Improvement (EI),
3) Lower Confidence Bound (LCB).

Bayesian optimization suits expensive black-box functions for two main reasons:

1) It does not require knowing the properties of the function (suitable for black-box functions),
2) It does not require a large number of evaluations (suitable for expensive functions).

F. System Modules

As shown in figure 1, our system *GraphOpt* is composed of four main components.

1) The Utility Module: it contains the shared information and configurations between all the algorithms. For example, the list of the parameters, the repeating factor, time limits and job execution commands. This module guarantees that all the experiments are run under the same circumstances.
2) The Job Runner: it contains the script which parses the chosen values of the parameters into an executable command. It runs the job and computes its execution time and then returns it to the optimization script.
3) The Optimization Module: it includes four optimization scripts:
   - Bayesian optimization,
   - Hill-climbing-based search algorithm,
   - Coordinate Descent,
   - Random Search.
4) The Graph Processing Framework module, which is composed of one master node and three workers. The computation cluster is created using Google Kubernetes Engine.

V. EXPERIMENTS AND RESULTS

In this section, we will present the setup of our experiments along with the results. We will start with two basic experiments. The first experiment is related to the procedure of selecting the parameters to tune. The second is related to the noise resulting from the nature of the cloud . After presenting the basic experiments, we will define a criterion that will be

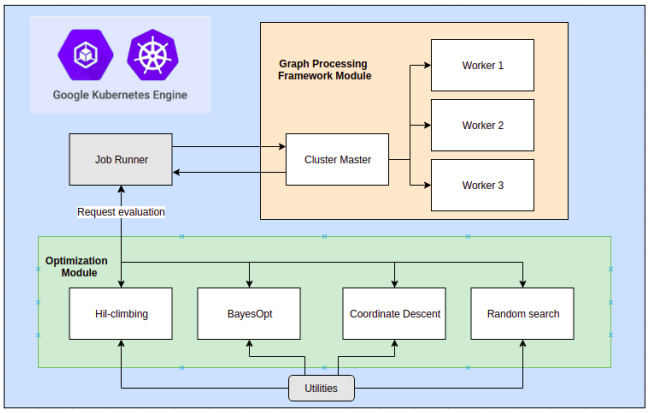

Fig. 1. Modules of GraphOpt

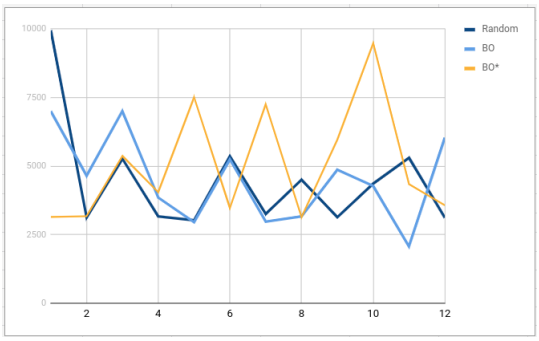

Fig. 2. The effect of removing parameters with a noisy effect

used to compare different methods. Then we will present the used datasets. Finally, we will present detailed experiments on multiple datasets, multiple graph algorithms, and multiple graph frameworks.

A. Basic Experiments

One of the problems we faced at the start of the project is choosing the set of parameters that we will optimize. The initial set of Giraph parameters we obtained from a recommendation by one the main developers of the framework. The list included 11 parameters as show in table I. We made a try to reduce this set. The idea is to measure the effect of each parameter independently and remove parameters with a noisy effect. A parameter is considered to have a noisy effect if the difference between the best and the worst performance obtained by changing the parameter is less than a noise threshold. The noise threshold we set is 2% of the average execution time of the job. This value was chosen empirically due to the fact that it is hard to tune. In section VI we describe our intention to tune it in future work. After removing the *parameters with a noisy effect*, only six parameters remained. This approach assumes that there is no second order -or higher- effect between the parameters. This is a strong assumption. To verify this choice, we compare the results between running the optimization algorithm for the original set of parameters and the reduced set. Figure 2 shows the effect of this step. The yellow curve represents the convergence of Bayesian optimization after removing the *parameters with a noisy effect*, the light-blue curve represents the convergence of Bayesian optimization using the original list of parameters, and the dark-blue curve represents the convergence of Random Search using the original list of parameters. The horizontal axis represents the number of optimization iterations (the number of evaluations of the target function), and the vertical axis represents the execution time. The graph job used is a *Page Rank* job and the framework used is Giraph. Clearly, the assumption behind this approach is false. For this reason, we kept the original set of parameters for the next experiments.

On a different matter, as shown in the state of the art [2], working in a distributed environment results in noisy measurement: running the same task with the same parameters values could result in different execution times. A common solution to this problem is to run the job multiple times and take the average value of the execution time. To verify this claim, we conducted two experiments to estimate this effect. Figure 3 shows a distribution of the difference between the worst executing time T_w and the best executing time T_b obtained by running the same job using the same values of the parameters (same configuration) for both GraphX and Giraph. As we can see, the difference between the best and the worst performance obtained by the same configuration could be up to 40% of the average execution time for Giraph jobs.

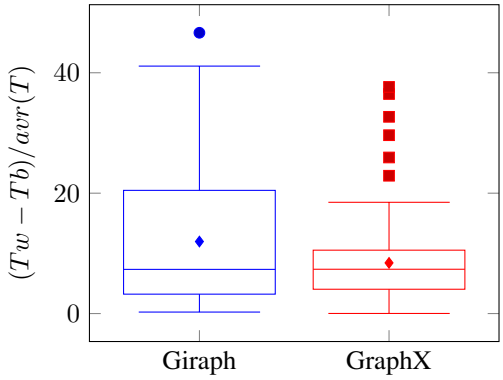

Fig. 3. The effect of noisy measurements

To explain the choice of running the same job multiple times and returning the average execution time, we run Bayesian optimization twice using the same graph job. Once with a repeat factor of one (we run the job only once under the same configuration), and another with a repeat factor of three. Figures 4 show the convergence of the two runs compared to random search. The horizontal axis represents the number of optimization iterations (the number of evaluations of the target function), and the vertical axis represents the execution time. The graph job used is a *Page Rank* job and the framework used is Giraph. When setting the repeat factor to 1, the behaviour of Bayesian Optimization (the curve in light-blue) is very similar to random search (the curve in dark-blue). However, when setting the repeat factor to 3 (the curve in yellow), it is clear

that the algorithm is converging. Our understanding of these results is that when providing a more accurate measurement to the optimization algorithm, we will converge faster.

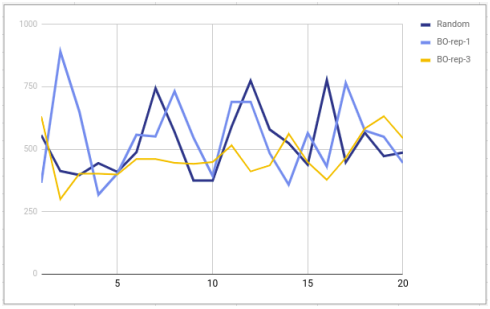

Fig. 4. The effect of noisy measurements.

B. Comparison Criteria

A basic loss function in our case could be defined as the difference between the execution time t' of the best configuration c' found by each algorithm and the execution time t^* of the optimal configuration c^*.

$$loss = t' - t^*$$

However, we do not know the optimal configuration c^*: it is not feasible to compute c^* in a reasonable time. An alternative loss function which is used in the state of the art [4], is the Optimality Gap. It is defined as the difference between the execution time t^b of the best configuration c^b obtained using a baseline optimization algorithm and execution time of the best configuration obtained using the optimization algorithm.

$$loss' = t' - t^b$$

In our experiments, we use a new utility function u to compare different optimization algorithms. The function measures, for a specific optimization algorithm A, how much better is it than a baseline algorithm.

$$u(A) = (t^b - t_A)/min(t^b, t_A)$$

Where t_A is the execution time of the best configuration c_A found by algorithm A. This function does not represent a loss but an improvement. The baseline algorithm that has been used in Random Search.

C. Datasets

For our experiments, we used multiple datasets. Some of these datasets are taken from *Graphalytics* [19], which contains multiple graph benchmarks. Others are taken from Stanford Network Analysis Project (SNAP) [20]. SNAP contains a set of real-life graphs including graphs representing social networks, road networks, and web pages. The complete list of used datasets are described in table III.

TABLE III
THE LIST OF GRAPH DATA SETS USED IN OUR EXPERIMENTS

Dataset Name	#Vertices	#Edges	Source
datagen-8_2-zf	43734497	106440188	*Graphalytics*
com-Orkut	3072441	117185083	SNAP
web-Goolge	875713	5105039	SNAP
com-youtube.ungraph	1134890	2987624	SNAP

D. Detailed Experiments

In this section, we will present the performance of Bayesian optimization, Hill-Climbing, and Coordinate Descent compared to random search.

1) Experimentation Setup: The master machine of the computation cluster is equipped with 4 CPU cores of type Intel(R) Xeon(R) @ 3.5GHz machine with 15 GB RAM running under Linux Ubuntu. Each of the 3 worker machines of the commutation cluster is equipped with 4 CPU cores of type Intel(R) Xeon(R) @ 3.5GHz machine with 26 GB RAM, running under Linux Ubuntu. All the machines are connected to the same local area network. We run each optimization algorithm 10 times (with different initialization points) and we present a box-plot representing a distribution of results (value of the utility function defined in V-B). Each figure in the following sections contains the described box-plot for Bayesian Optimization (BO), Hill-Climbing (HC) search, and Coordinate Descent (CD). We run the Coordinate Descent algorithm only once since it is a deterministic algorithm. In this case the box-plot of CD represents the result of CD compared to 10 runs of Random Search.

Regarding our implementation of a Hill-Climbing-based search algorithm 1, the number of initializing points m we used is 10 and the value of k is 3 with a total number of evaluation equals to 20. Regarding our implementation of Bayesian Optimization we initialized with 10 iterations with a total number of evaluations equals to 20. The number of evaluations used for one run of random search is equal to 20. The experiments use multiple graph algorithms as the graph job. We present the results for both GraphX and Giraph.

2) Results: In this section we present the results. Since the implemented algorithms are not deterministic and sensitive to initialization points, these plots (5 to 12) display a distribution of the obtained values of utility function. First, will we present the results we obtained with Giraph followed by the results obtained with GraphX.

Giraph

Figures 5, 6, 7, and 8 show box-plots of the utility function u for different graph jobs implemented in Giraph.

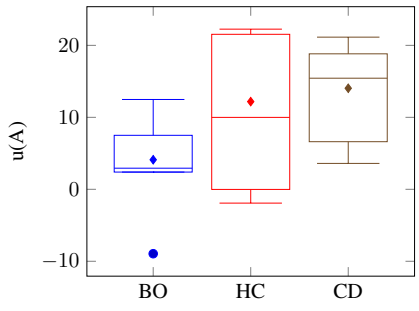

Fig. 5. Using a Page Rank job

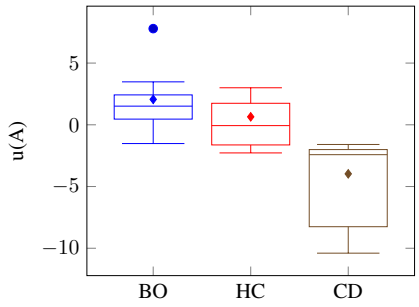

Fig. 6. Using a Single Source Shortest path job

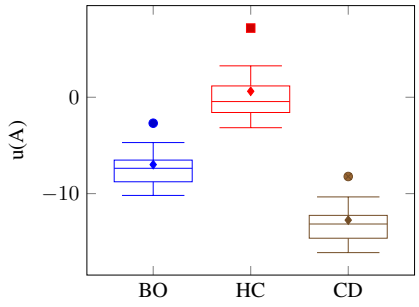

Fig. 7. Using a Breadth First Search job

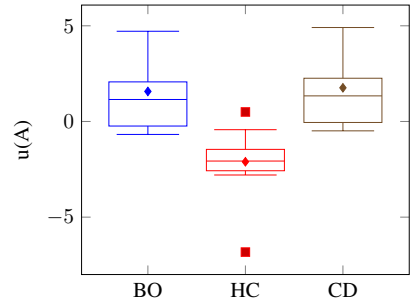

Fig. 8. Using a Weakly Connected Component job

GraphX

Figures 9, 10, 11, and 12 shows box-plots of the utility function u for different graph jobs implemented in GraphX.

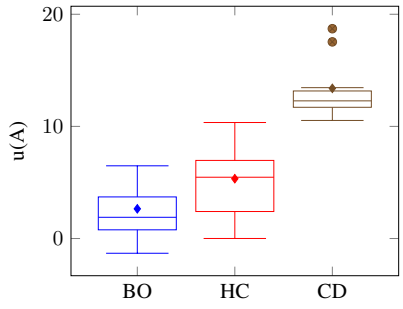

Fig. 9. Using a Page Rank job

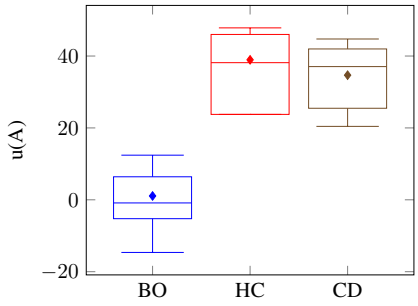

Fig. 10. Using a Single Source Shortest path job

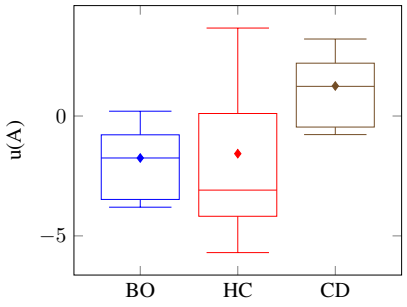

Fig. 11. Using a Breadth First Search job

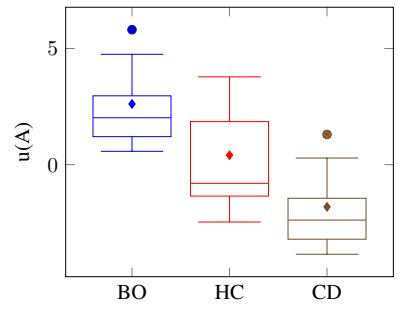

Fig. 12. Using a Weakly Connected Component job

3) Discussion: After reviewing the previous results we notice that different graph jobs reacted differently to the optimization techniques. In most cases, two of the optimization techniques were able to find better configurations than random search (on average). The worst case for both Giraph and GraphX frameworks was using a BFS job. This was the only case where only one technique was able to outperform random search (on average). To summarize the magnitude of the improvement, we present figure 13. This figure shows the achieved improvement of each optimization algorithm considering all the graph jobs that we used for experiments. The average improvement for the Bayesian Optimization technique is only 0.6%. The poor improvement under a BFS job has eliminated all the positive improvement under the other jobs. Regarding the Hill-climbing approach, the average improvement equals to 4.4% with a maximum improvement of more than 47.8%. We can consider this technique as the best one in our system: the maximum achieved improvement is very high with an *acceptable* worst case of -6.8%. Finally, the Coordinate Descent approach achieved the most unstable performance. With a maximum improvement of 44.7%, an average improvement of 6%, and a worst performance of -16.1%, we do not recommend to use this approach.

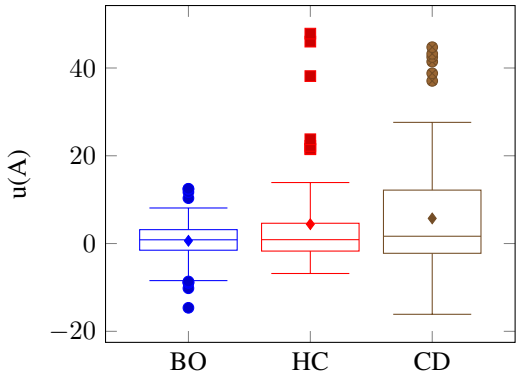

Fig. 13. Summary of the results

VI. Conclusion

In this paper, we introduced *GraphOpt*: an automatic black-box parameter tuning framework. This framework uses optimization techniques to find the best configuration for graph processing frameworks. *GraphOpt* includes a new implementation of a Hill-Climbing-based search algorithm. The experiments we carried out show that Bayesian Optimization, Hill-climbing search, and Coordinate Descent are able to find better configurations compared to random search. A future milestone for the project would be to focus on the strategy of choosing initialization points. Improving this strategy could result in a great improvement due to the high sensitivity of Bayesian Optimization and Hill-climbing search to the choice of the initial points.

As a next step for *GraphOpt*, we intend to support new graph processing frameworks and new optimization techniques and to tune the value of noise threshold that was chosen empirically to suit a large diversity of jobs and workloads. Moreover, we are going to merge our work with the work of another research track at the company. This track focuses on building a generic prediction model for the performance of Spark jobs. Including this model with our core unit could accelerate the search process.

Acknowledgment

We would like to thank Luc Delepine for his feedback and also Florian Demesmaeker and Gil Degrove for their technical support.

References

[1] M. Bilal and M. Canini, "Towards automatic parameter tuning of stream processing systems," in *Proceedings of the 2017 Symposium on Cloud Computing*, pp. 189–200, ACM, 2017.

[2] L. Fischer, S. Gao, and A. Bernstein, "Machines tuning machines: Configuring distributed stream processors with bayesian optimization," in *2015 IEEE International Conference on Cluster Computing*, pp. 22–31, IEEE, 2015.

[3] O. Alipourfard, H. H. Liu, J. Chen, S. Venkataraman, M. Yu, and M. Zhang, "Cherrypick: Adaptively unearthing the best cloud configurations for big data analytics," in *14th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 17)*, pp. 469–482, 2017.

[4] D. Golovin, B. Solnik, S. Moitra, G. Kochanski, J. Karro, and D. Sculley, "Google vizier: A service for black-box optimization," in *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pp. 1487–1495, ACM, 2017.

[5] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica, "Graphx: Graph processing in a distributed dataflow framework," in *11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14)*, pp. 599–613, 2014.

[6] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein, "Distributed graphlab: a framework for machine learning and data mining in the cloud," *Proceedings of the VLDB Endowment*, vol. 5, no. 8, pp. 716–727, 2012.

[7] N. Sundaram, N. Satish, M. M. A. Patwary, S. R. Dulloor, M. J. Anderson, S. G. Vadlamudi, D. Das, and P. Dubey, "Graphmat: High performance graph analytics made productive," *Proceedings of the VLDB Endowment*, vol. 8, no. 11, pp. 1214–1225, 2015.

[8] A. Roy, I. Mihailovic, and W. Zwaenepoel, "X-stream: Edge-centric graph processing using streaming partitions," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pp. 472–488, ACM, 2013.

[9] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "Powergraph: Distributed graph-parallel computation on natural graphs," in *Presented as part of the 10th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 12)*, pp. 17–30, 2012.

[10] R. R. McCune, T. Weninger, and G. Madey, "Thinking like a vertex: a survey of vertex-centric frameworks for large-scale distributed graph processing," *ACM Computing Surveys (CSUR)*, vol. 48, no. 2, p. 25, 2015.

[11] A. G. v. L. Kenji Okada1, Marcos Amars Gonzalez1, "Scheduling moldable bsp tasks on cloudsthiago,"

[12] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: a system for large-scale graph processing," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pp. 135–146, ACM, 2010.

[13] M. D. Morris and T. J. Mitchell, "Exploratory designs for computational experiments," *Journal of statistical planning and inference*, vol. 43, no. 3, pp. 381–402, 1995.

[14] C.-J. Hsu, V. Nair, T. Menzies, and V. W. Freeh, "Scout: An experienced guide to find the best cloud configuration," *arXiv preprint arXiv:1803.01296*, 2018.

[15] B. Shahriari, K. Swersky, Z. Wang, R. P. Adams, and N. De Freitas, "Taking the human out of the loop: A review of bayesian optimization," *Proceedings of the IEEE*, vol. 104, no. 1, pp. 148–175, 2015.

[16] M. Li, L. Zeng, S. Meng, J. Tan, L. Zhang, A. R. Butt, and N. Fuller, "Mronline: Mapreduce online performance tuning," in *Proceedings of the 23rd international symposium on High-performance parallel and distributed computing*, pp. 165–176, ACM, 2014.

[17] B. Xi, Z. Liu, M. Raghavachari, C. H. Xia, and L. Zhang, "A smart hill-climbing algorithm for application server configuration," in *Proceedings of the 13th international conference on World Wide Web*, pp. 287–296, ACM, 2004.

[18] T. Ye, S. Kalyanaraman, "A recursive random search algorithm for large-scale network parameter configuration," in *ACM SIGMETRICS Performance Evaluation Review*, pp. 196–205, ACM, 2004.

[19] A. Iosup, T. Hegeman, W.L. Ngai, S. Heldens, A. Prat-Prez, T. Manhardto, H. Chafio, M. Capot, N. Sundaram, M. Anderson, and I.G. Tnase, "LDBC Graphalytics: A benchmark for large-scale graph analysis on parallel and distributed platforms" in *Proceedings of the VLDB Endowment*, pp. 1317-1328, ACM, 2016.

[20] M. Zitnik, R. Sosič, S. Maheshwari, and J. Leskovec, "BioSNAP Datasets: Stanford Biomedical Network Dataset Collection,", 2018.