École Polytechnique

BACHELOR THESIS IN COMPUTER SCIENCE

Extending MOO methods for stage-level ressource optimization

Author:

Philippe Guyard, École Polytechnique

Advisor:

Yanlei Diao, Cedar Lab, Inria Saclay

Academic year 2022/2023

Abstract

Cloud data analytics at the production scale presents a highly complex environment for resource optimization, a problem crucial for meeting performance goals and budgetary constraints of analytical users. Our recent research [[4],[7]] (via joint work with Amazon AWS and Alibaba Cloud) brings principled multi-objective optimization and large-scale machine learning to bear on the process of performance and cost modeling and thereby enable intelligent resource optimization decisions based on such models. We envision that such techniques will be broadly applied to a range of cloud products, unlock the power of big data and machine learning for intelligent decision making, and enable big leaps forward for offering the best cost-performance benefits to real-world users and ultimately, long-term sustainable computing.

In this project, we consider the problem of multi-objective optimization (MOO) over a large query graph. In big data analytics systems like Spark, a query is modeled as a direct acyclic graph (DAG) of stages, where a stage is a DAG of operators itself. In prior work [[4],[7]], we have developed MOO algorithms to find Pareto-optimal solutions for each stage. We would like to combine stage-level Pareto-optimal solutions to derive global Pareto-optimal solutions for the entire query graph.

In this paper, we first formalize our problem, then present an algorithm to efficiently solve some particular cases and give a good approximation of the result for the remaining cases. The core of our work is a novel Divide-And-Conquer approach to computing Pareto Frontier of computational graphs. Detailed experiments using benchmark workloads show that our algorithm provides an exponential speedup over the naive approach, while offering a good coverage of the true Pareto Frontier. Compared to the real Pareto Frontier, our algorithm yields approximations that are at least 60% as good, but can be computed thousands of times faster. For some cases (which cover the majority of production graphs), we also offer formal proofs of correctness of our algorithms and detailed complexity analysis.

Contents

1 Introduction

As the volume of data generated by enterprises has continued to grow, big data analytics in the cloud has become commonplace for obtaining business insights from this voluminous data. Despite its wide adoption, current big data analytics systems such as Spark remain best effort in nature and typically lack the ability to take user objectives such as performance goals or cost constraints into account.
Determining an optimal hardware and software configuration for a big-data analytic task based on user-specified objectives is a complex task and one that is largely performed manually. Consider an enterprise user who wishes to run a mix of Spark analytic tasks in the cloud. First, she needs to choose the server hardware configuration from the set of available choices, e.g., from over 190 hardware configurations offered by Amazon's EC2 [1]. These configurations differ in the number of cores, RAM size, availability of solid state disks, etc. After determining the hardware configuration, the user also needs to determine the software configuration by choosing various runtime parameters. For the Spark platform, these runtime parameters include the *number of executors*, *cores per executor*, *memory per executor*, *parallelism* (for reduce-style transformations), *Rdd compression* (boolean), *Memory fraction* (of heap space), to name a few.
The choice of a configuration is further complicated by the need to optimize *multiple*, possibly conflicting, user objectives. Consider the following real-world use cases at data analytics companies and cloud providers that elaborate on these challenges and motivate our work:
Use Case 1 (Data-driven Business Users). A data-driven security company that runs thousands of cloud analytic tasks daily has two objectives: keep the latency low in order to quickly detect fraudulent behaviors and also reduce cloud costs that impose substantial operational expenses on the company. For cloud analytics, task latency can often be reduced by allocating more resources, but at the expense of higher cloud costs. Thus, the engineers face the challenge of deciding the cloud configuration and other runtime parameters that balance latency and cost.
Use Case 2 (Serverless Analytics). Cloud providers now offer databases and Spark for serverless computing [2], [5]. In this case, a database or Spark instance is turned off during idle periods, dynamically turned on when new queries arrive, and scaled up or down as the load changes over time. For example, a media company that uses a serverless offering to run a news site sees peak loads in the morning or as news stories break, and a lighter load at other times. The application specifies the minimum and maximum number of computing units (CUs) to service its workload across peak and off-peak periods; it prefers to minimize cost when the load is light and expects the cloud provider to dynamically scale CUs for the morning peak or breaking news. In this case, the cloud provider needs to balance between latency under different data loads and user cost, which directly depends on the number of CUs used.
Overall, choosing a configuration that balances multiple conflicting objectives is non-trivial—even expert engineers are often unable to choose between two cluster options for a single objective like latency [6], let alone choosing between dozens of cluster options for multiple competing objectives.
In this project, we consider the problem of multi-objective optimization (MOO) over a large Spark query graph. In big data analytics systems like Spark, a query is modeled as a direct acyclic graph (DAG) of stages, where a stage is a DAG of operators itself. In prior work [[4],[7]], we have developed MOO algorithms to find Pareto-optimal solutions for each stage. In this paper, we would like to combine stage-level Pareto-optimal solutions to derive global Pareto-optimal solutions for the entire query graph. First, we introduce a naive algorithm that works in exponential time. While this is useful for running benchmarks, the naive algorithm is unfortunately unusable in practice due to our time constraints. Indeed, we want the total running time to be less than a few seconds, while the naive algorithm can easily take days or years to complete. Next, we will explore particular graph structures for which our task can be solved efficiently. Finally, we will combine the first two algorithms to achieve good approximation capabilities for any query graph.

2 Problem statement

2.1 Looking at a particular example

Let us imagine that we have a very long-running task that we want to execute on our local cluster (that is also busy with other tasks) with Apache Spark. For example, we have a list of millions of books that we store on a local hard drive, and we want to make sure that their contents are equivalent to what can be found online. Note that this is a toy example that we use to illustrate our problem, and hence many simplifications will be made. To execute this task, we break it down to the following steps:

1. Fetch the online versions of the books from some website

2. Fetch our own versions from our hard drive
3. Compute hashes of the online books' content
4. Compute hashes of the local books' content
5. Compare the hashes and return any possible differences

Observe that we have some requirements on the order of executions of the above steps. For instance, steps 1 and 2 can be executed in parallel, step 3 has to follow step 1, step 4 has to follow step 2 and step 5 has to come last. These complex dependencies can be encoded by a *dependency graph*, as shown in Figure 1.
This dependency graph is how our Spark query will look like. After feeding it to the UDAO system (from [7]), we will get different configurations and resulting objectives for every node in this graph. For simplicity, let us assume that UDAO returned two configurations for every node, such that

- Configuration 1 is one that provides the lowest possible latency, but requires a lot of resources, and hence the cost of running it on our cluster is high
- Configuration 2 is one that provides the highest possible latency, but requires little resources, and hence the cost of running it on our cluster is low

Even in this simple example, there are many configurations that one can choose for our task. For example, we could

- execute both fetches fast with high cost, then execute both hash computations slowly with low cost, then execute the comparison fast
- or execute everything slowly with low cost
- or execute the local part of the graph fast, and the rest slowly
- or . . .

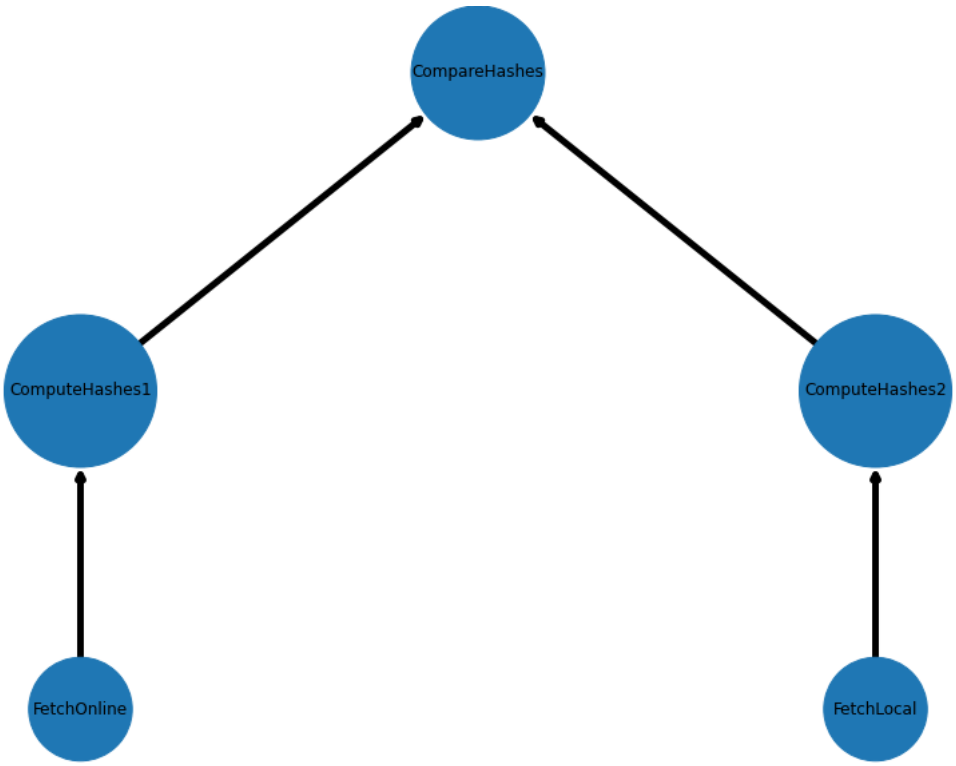


Figure 1: Visualization of our example task

However, out of all these configurations, only a few makes sense. Indeed, one can imagine that the "Fetch Online" node will have much higher latencies than the rest of the nodes, simply because it needs to download millions of books from the internet. In this context, we know that, no matter how slowly we execute the "Fetch Local + Compute Hashes 2" portion of the graph, the bottleneck of the computation will always be the "Fetch Online" node. As such, it makes sense to allocate a lot of resources to "Fetch online + Compute Hashes 1" (so choose Configuration 1 for these two nodes), while choosing Configuration 2 for nodes "Fetch Local + Compute Hashes 2". The configuration for "Compare Hashes" could be anything.
This example illustrates exactly the task we will trying to solve in this paper. Given a Spark Query and a Pareto optimal list of configurations for each node (i.e each stage), find configurations that are optimal for the whole graph. We call the list of all optimal configurations for a graph its Pareto Frontier.

2.2 Assumptions

In this paper, we restrict ourselves to a simpler version of the original task, by introducing two key assumptions:

Assumption 1. *For every stage, we can choose any of its configuration without affecting other stages.*

Assumption 2. *The input Pareto Frontiers of stages are constant, i.e do not change with time or state of the cluster.*

We will also add two points that are not strictly necessary for the algorithms nor the proofs, but it will make the explanations much easier.

- The Pareto Frontiers of stages are all sets of points in R^2, representing a pair of (latency, cost) objectives. The latency being how fast a node will complete execution, and the cost representing how much total cluster resources would need to be used.
- The Pareto fronts of all nodes are of the same size $k \in \mathbb{N}^*$

Note that the two points above are trivial to drop for all presented algorithms, and are introduced just to reduce the confusion with notations.

2.3 Graph theory statement

With the two assumptions in mind, we can formalize the problem statement.
Our input is a Spark job, which can be represented as a Directed acyclic graph $G(V, E)$ of stages. Each node $v \in V$ is associated with the Pareto frontier $\mathcal{F}_v$ of some stage of the Spark query computed in advance. Each point of the Pareto Frontier also stems from a specific resource configuration θ_v. From Assumption 1, we know that we can choose these resource configurations independently for each node. More specifically, we have

$$\mathcal{F}_v = \{F_1^v, \dots, F_k^v\},\ F_m^v = (f_1, f_2) \in \mathbb{R}^2,\ \forall 1 \leq m \leq k$$

The task is to find a function

$$\omega : v \in V \mapsto (f_1, f_2) \in \mathcal{F}_v,$$

which produces a solution that is Pareto optimal for the entire graph G. In other words, we want to choose one of the k configurations of each node, such that the overall running time and cost of the graph G is Pareto Optimal. By taking the resource configuration θ_v from each node and merging them together, we thus create a Pareto Optimal resource configuration θ_G for the entire graph, that we will later show as a recommendation to the user. We define the notion of Pareto optimality in subsection 2.4.

2.4 Defining the Pareto Front of a DAG

Assume that we have some Spark query $G = (V, E)$ and some configuration ω on this query. That is, for each node, we have chosen exactly one (latency, cost) pair. With this, we can compute the total latency and cost of 'running' G.
By calling Algorithm 1 at the root r of the graph G and Algorithm 2, we can compute the objectives of G under ω. We will define

$$F_G^\omega = (\text{LatencyAt}(r(G), \omega), \text{CostOf}(G, \omega)$$

to be the objective associated to ω.
From this, naturally follows the definition of objective space. We define Φ_G the objective space of G as

$$\Phi_G = \{F_G^\omega, \forall \omega \text{ configurations of } G\}.$$

We say that F_G^ω Pareto dominates another objective $F_G^{\omega'} \in \Phi_G$ if and only if

$$(F_G^\omega)_i \leq (F_G^{\omega'})_i, \forall 1 \leq i \leq 2 \text{ and } \exists j \in [1,2], (F_G^\omega)_j < (F_G^{\omega'})_j$$

Finally, we can now define the Pareto-optimality of an objective F_G^ω:

F_G^ω is Pareto optimal if and only if there does not exist $F_G^{\omega'} \in \Phi_G$ such that $F_G^{\omega'}$ Pareto dominates F_G^ω

NOTE: We call Algorithm 1 and Algorithm 2 *accumulators* for latency and cost respectively. To generalize our approach to metrics other than latency and/or cost, one needs to define accumulators of these metrics for the entire graph.

Algorithm 1 LatencyAt(v, ω)

if v is a leaf **then**
  Just return the latency of v under ω
  return $\omega(v)_1$
else
  $maxChild \leftarrow 0$
  for $c \in children(v)$ **do**
    $maxChild \leftarrow \max(maxChild, LatencyAt(c, \omega))$
  end for
  return $\omega(v)_1 + maxChild$
end if

Algorithm 2 CostOf(G, ω)

$c \leftarrow 0$
for $v \in V_G$ **do**
  $c \leftarrow c + \omega(v)_2$
end for
return c

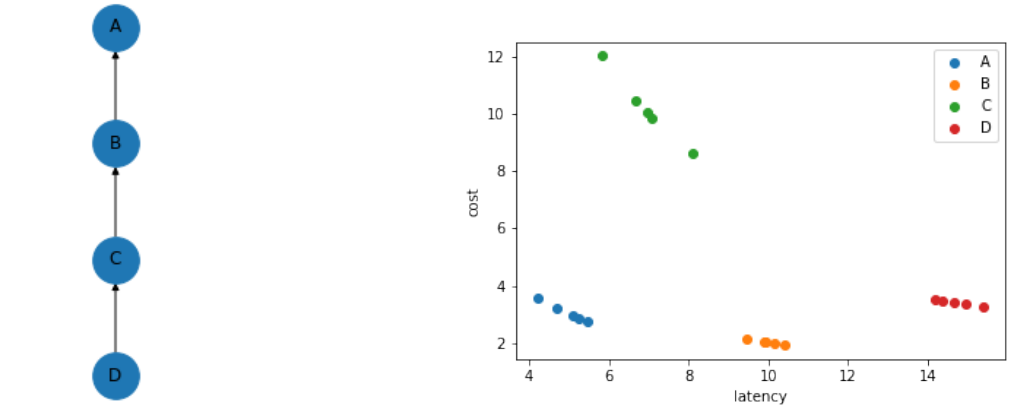
3 Naive approach

In this section, we propose a naive way of solving our task. One can simply iterate through all possible configurations of G, put the resulting points in an array, and compute its Pareto frontier. An implementation of this idea can be found in Algorithm 3.

Algorithm 3 NaiveParetoFronDag(G)

$r \leftarrow$ root of G
$\Omega \leftarrow$ all possible configurations of G
allPoints $\leftarrow \emptyset$
for $\omega \in \Omega$ **do**
  $f_1 \leftarrow$ LatencyAt(r, ω)
  $f_2 \leftarrow$ CostAt(r, ω)
  Add (f_1, f_2) to allPoints
end for
return Pareto Front of allPoints

There are two issues with Algorithm 3 that make in unusable in production:



(a) Linked list example

(b) Pareto frontiers of each node

Figure 2: Example of linked list structure

1. The complexity of computing Ω grows exponentially with the number of nodes $n = |V|$. In fact, we can think of a configuration ω of G as a k-coloring of G, since we associate every vertex $v \in V$ with one of the k points in its Pareto Frontier. Since Ω is then basically the set of all k-colorings of G, we know that $|\Omega| = k^n$, and hence the complexity of the loop in Algorithm 3 is $\Theta(k^n)$.

2. Even in the small number of cases where n and k are small enough that an $\Theta(k^n)$ can finish under our time constraints, the computation of the Pareto Front is quadratic in the number of points, and the actual runtime of Algorithm 3 is $\Theta(k^{2n})$, which does not satisfy our time constraints in most meaningful cases

4 Computing the Pareto Frontier of specific graph structures

In this section, we simply define different graph structures for which their Pareto Frontier can be computed very efficiently and present algorithms to compute them. All proofs of correctness, as well as complexity analysis, can be found in section 7.

4.1 'Linked List' structures

We will now present an algorithm to compute the Pareto frontier of a specific graph structure: the linked list (see Figure 2 for an example).

We want to apply a Divide-And-Conquer approach to computing the Pareto Front of G. First, we need to define a merge operation, which takes in the Pareto Fronts two 'Linked Lists' H and G, and computes the Pareto Front of their 'Joint' list (achieved by a concatenation of H and G into one list). This can be done with Algorithm 4.

Observe that, on the last line of Algorithm 4, there is a call to $ComputeParetoFrontier$. We assume that this is a method available to us through some library function, that returns the pareto front of a list of points. We will not provide an implementation for it, as it is outside the scope of the paper. Throughout the paper, we will assume the complexity of this method to be $O(n^2)$, where n is the size of the input list of points.

Algorithm 4 ComputeSequentialFrontier($\mathcal{F}_H, \mathcal{F}_G$)

$allPoints \leftarrow \emptyset$
for $(f_1, f_2) \in \mathcal{F}_H$ **do**
  for $(f'_1, f'_2) \in \mathcal{F}_G$ **do**
    $allPoints \leftarrow allPoints \cup \{(f_1 + f'_1, f_2 + f'_2)\}$
  end for
end for
return $ComputeParetoFrontier(allPoints)$

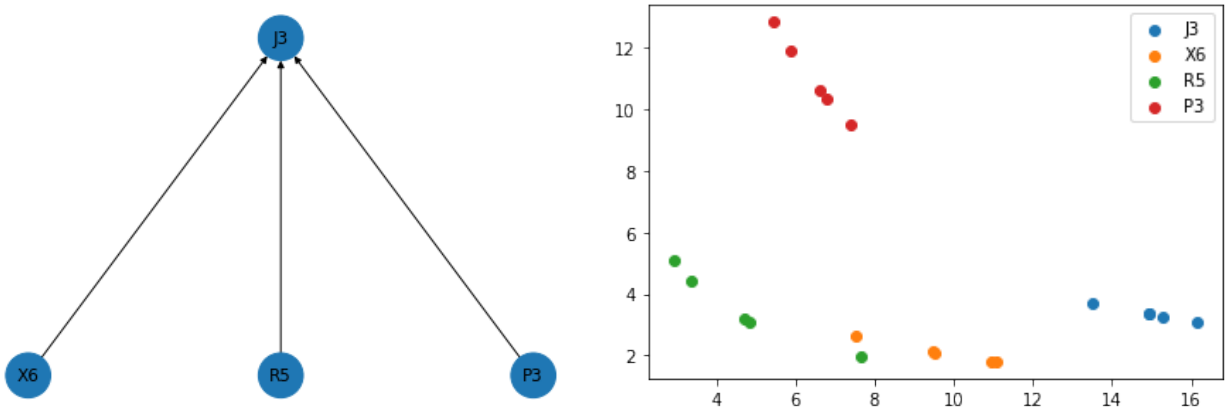
For this next step, we can think of our input graph as a simple array of vertices $\mathcal{A}$, where the first element is the root, the second element is the child of the root, etc. For example, in Figure 2, we would have $\mathcal{A} = [A, B, C, D]$. Now, given any subarray $\mathcal{S} \subseteq \mathcal{A}$, we can use Algorithm 5 to compute it's Pareto Front.

Algorithm 5 SequentialDivideAndConquer($\mathcal{S}$)

if $|\mathcal{S}| = 1$ **then**
  return $\mathcal{S}[0]$
else
  $H \leftarrow$ first half of $\mathcal{S}$
  $G \leftarrow$ second half of $\mathcal{S}$
  $\mathcal{F}_H \leftarrow SequentialDivideAndConquer(H)$
  $\mathcal{F}_G \leftarrow SequentialDivideAndConquer(G)$
  return $ComputeSequentialFrontier(\mathcal{F}_H, \mathcal{F}_G)$
end if

Computing the frontier of a 'Linked list' now simply reduces to calling $SequentialDivideAndConquer(\mathcal{A})$

4.2 'Join' structures



(a) 'Join' example

(b) Pareto frontiers of each node

Figure 3: Example of 'Join' structure

In this section, we will present an algorithm to compute the Pareto frontier of 'Join' structures (see Figure 3 for an example). We can apply a similar trick to subsection 4.1. We think of a 'Join' as a pair $(p, \mathcal{C})$, where p is the parent node, and $\mathcal{C}$ an array of children, written in any order. For example, in Figure 2, the parent is the node J3, while the children could be any of $\mathcal{C} = [X6, R5, P3], [X6, P3, R5], \ldots$

First, we need to apply the Divide-And-Conquer idea to the list of children. We will proceed in exactly the same way as subsection 4.1. The only thing we need to do is slightly modify the 'merge' phase of Divide-And-Conquer. Indeed, in a 'Join', all children phases can be run at the same time, and hence the total latency of running the list of children is the maximum latency between all children. This change is reflected in Algorithm 6.

Algorithm 6 ComputeParallelFrontier($\mathcal{F}_H, \mathcal{F}_G$)

$allPoints \leftarrow \emptyset$
for $(f_1, f_2) \in \mathcal{F}_H$ **do**
  for $(f'_1, f'_2) \in \mathcal{F}_G$ **do**
    $allPoints \leftarrow allPoints \cup \{(\max(f_1, f'_1), f_2 + f'_2)\}$
  end for
end for
return $ComputeParetoFrontier(allPoints)$

Algorithm 7 ParallelDivideAndConquer($\mathcal{C}$)

if $|\mathcal{C}| = 1$ **then**
  return $\mathcal{C}[0]$
else
  $H \leftarrow$ first half of $\mathcal{C}$
  $G \leftarrow$ second half of $\mathcal{C}$
  $\mathcal{F}_H \leftarrow ParallelDivideAndConquer(\mathcal{H})$
  $\mathcal{F}_G \leftarrow ParallelDivideAndConquer(G)$
  return $ComputeParallelFrontier(\mathcal{F}_H, \mathcal{F}_G)$
end if

Now we can use Algorithm 7 to compute the Pareto front of all children. With this, we can replace the array of children $\mathcal{C}$ by an artificial node c that has a frontier $\mathcal{F}_c = ParallelDivideAndConquer(\mathcal{C})$. With that, our join becomes a simple list of two nodes, and we can use algorithms from subsection 4.1 to compute its Pareto frontier. This is implemented in Algorithm 8.

Algorithm 8 ParetoFrontJoin($p, \mathcal{C}$)

$\mathcal{F}_c \leftarrow ParallelDivideAndConquer(\mathcal{C})$
return $ComputeSequentialFrontier(\mathcal{F}_p, \mathcal{F}_c)$

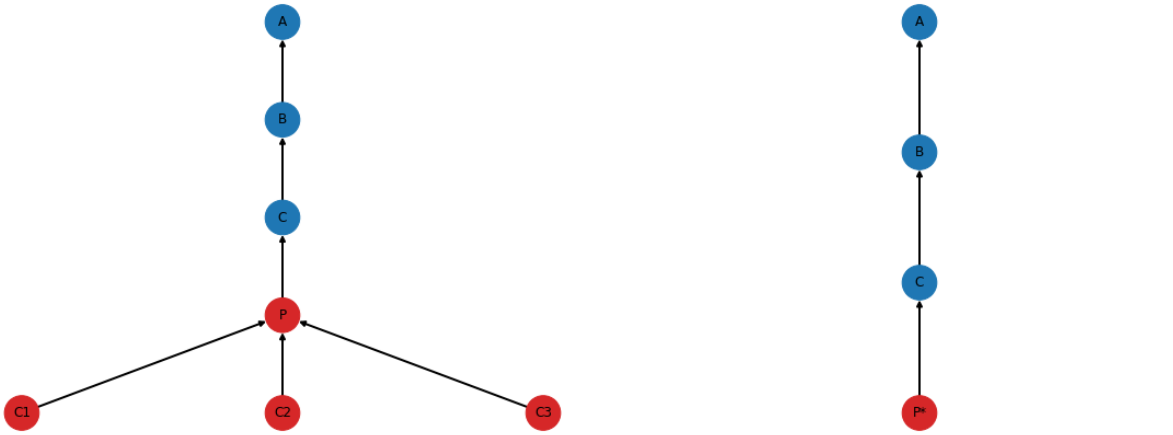
4.3 A small note on compressing nodes

Observe that, in subsection 4.1 and subsection 4.2, we only care about the Pareto Frontiers underlying every node, abstracting away from the concepts of Apache Spark Stages. This is useful, as with that thinking, we can replace any sub-graph H of our query G by a single node v_H such that $\mathcal{F}_{v_H} = \mathcal{F}_H$. This is illustrated in Figure 4, where the node P is the parent of a 'Join'. The frontier of the join is computed with Algorithm 8, then put in the node $P*$, and G is modified accordingly

4.4 Trees

Observe that any tree can be seen as a combination of 'Join' and 'Linked List' structures. We already know how to compute Pareto fronts of these structures from subsection 4.1 and subsection 4.2. We can now proceed to generalize our approach to any tree. For that, we will recursively apply the following sequence of operations:

1. Find the "deepest" base structure in G (i.e 'Join' or 'Linked List')
2. Compress it (in the sense of subsection 4.3) to one node using the algorithms we described above (Algorithm 5 or Algorithm 8)
3. Replace this structure by the compressed node in G and repeat until G only has 1 node left
4. The last node will contain the Pareto Front of G



(a) Before compression: sub-graph H is marked in red

(b) After compression: P* contains the frontier of H

Figure 4: Example of compressing a join inside of a simple Tree

Algorithm 9 ParetoFrontTree(v)

if $isTerminalJoin(v)$ **then**
  return $ParetoFrontJoin(v, children(v))$
end if
if $isTerminalList(v)$ **then**
  return $SequentialDivideAndConquer(getList(v))$
end if
if $|children(v)| = 1$ **then**
  $u \leftarrow getEndOfList(v)$
  $u' \leftarrow ParetoFrontTree(u)$
  replace u by u' in G
  $\mathcal{S} \leftarrow makeList(v, u')$
  return $SequentialDivideAndConquer(\mathcal{S})$
else
  We have: $|children(v)| \geq 2$
  for $u \in children(v)$ **do**
    $u' \leftarrow ParetoFrontTree(u)$
    replace u by u' in G
  end for
  $children(v)$ now return the updated children of v
  return $ParetoFrontJoin(v, children(v))$
end if

Formally, this procedure can be described by the following algorithm. When we use the terminology of 'replacing' a node v by another node u, we refer to the operation described in subsection 4.3.

Observe that, in Algorithm 9, we use multiple utility functions, for which we do not provide implementations. We will provide here a brief description of what they do:

- $isTerminalJoin$(v) returns $True$ if and only if the subgraph of G rooted at v is a simple 'Join'
- $isTerminalList$(v) returns $True$ if and only if the subgraph of G rooted at v is a simple 'Linked List'
- $getEndOfList(v)$ returns the *first* node u *directly under* v for which $children(u) \geq 2$ holds.
  - By *first* we mean with the lowest depth in the tree G
  - By *directly under* we mean that u can be reached from v by following edges in G
- $makeList(head, tail)$ makes a list by including all the elements from $head$ to $tail$
  - For example, this can be done by recursively following the single dependency of $head$ until we reach $tail$

For reference, Figure 5 contains a useful visualization for how $ParetoFrontTree$ traverses a particular tree. We start at the root A of the tree, and traverse it in a Depth-First Search fashion. However, instead of looking at individual nodes, we traverse entire structures (more specifically Joins and Linked Lists).

5 Computing the Pareto Frontier of general DAGs

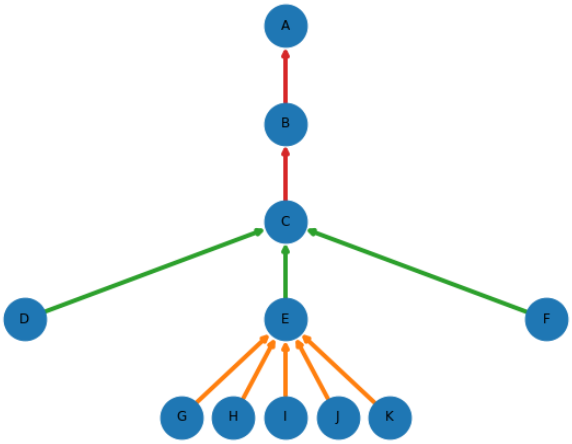
In this section, we will exclusively look at DAGs which are not trees, as the case of trees was covered in subsection 4.4. In other words, for all the graphs that we will be studying here, there exist at least one node that has multiple parents. Throughout the section, we will call such nodes 'splits'.

5.1 Removing unnecessary dependencies

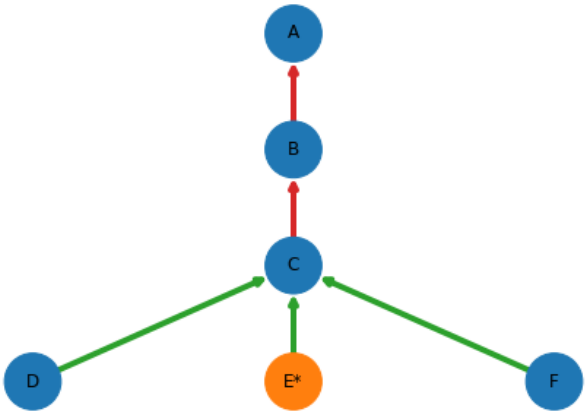
Let G be a DAG and let v be a split in G. We will now study the case where some outgoing edges of v may be removed from G without its Pareto Front. For that, we define the concepts of minimal and maximal latency of a node using Algorithm 10 and Algorithm 11 respectively.

Algorithm 10 MinLatencyAt(v)

$mmin \leftarrow \infty$
for $(f_1, f_2) \in \mathcal{F}_v$ **do**
  $mmin \leftarrow \min(f_1, mmin)$
end for
Now $mmin$ holds the smallest possible latency that the node v can take
if v is a leaf **then**
  return $mmin$
else
  $maxChild \leftarrow 0$
  for $c \in children(v)$ **do**
    $maxChild \leftarrow \max(maxChild,\ MinLatencyAt(c))$
  end for
  return $mmin + maxChild$
end if



(a) Step 1: The Join marked by the orange edges is a terminal Join



(b) Step 2: The orange Join was compressed into E*. Now the Join marked byy green edges is a terminal Join

(c) Step 3: The green Join was compressed into C*. Now the whole graph is just a list with head A and tail C*

(d) Step 4: The list was compressed. The frontier of A* is now the frontier of the graph

Figure 5: Visual representation of how $ParetoFrontTree$ computes the Pareto Frontier of a particular tree

Algorithm 11 MaxLatencyAt(v)

$mmax \leftarrow \infty$
for $(f_1, f_2) \in \mathcal{F}_v$ **do**
  $mmax \leftarrow \max(f_1, mmax)$
end for
Now $mmax$ holds the smallest possible latency that the node v can take
if v is a leaf **then**
  return $mmax$
else
  $maxChild \leftarrow 0$
  for $c \in children(v)$ **do**
    $maxChild \leftarrow \max(maxChild,\ MaxLatencyAt(c))$
  end for
  return $mmax + maxChild$
end if

Now consider the following scenario:

- $v \in V$ has multiple parents
- For some parent p, v has a sibling at p whose minimal latency is higher than the maximum latency of v
- Then we know that the latency of p will never be affected by v, since p will always have another child that needs more time
- The cost of v will also be accounted by other parents of v. Since p is not the only parent, we can thus safely remove the edge from v to p

This observation is formalized in Algorithm 12. A trivial example is shown in Figure 6. Indeed, if we set a, b and c to be the latencies of A, B and C respectively, we know that the latency at A will be $latencyAt(A) = a + max(c, latencyAt(B)) = a + max(c, c + b) = a + b + c$. Clearly, the edge C -> A can be removed.

Algorithm 12 RemoveUnnecessaryDependencies(G)

for $v \in splits(G)$ **do**
  for $p \in parents(v)$ **do**
    for $s \in children(v) \setminus \{v\}$ **do**
      if MinLatencyAt(s) $\geq$ MaxLatencyAt(v) **then**
        RemoveEdge(G, v, p)
        break the loop
      end if
    end for
    if $|parents(v)| = 1$ **then**
      break the loop
    end if
  end for
end for

5.2 Approximation techniques and Final Algorithm

We did not manage to solve our problem for the case of a general DAG. However, using the techniques described in previous sections, one can efficiently compute good approximations of the Pareto Front.
Let G be a general DAG. We first apply the following transformations to G:

1. Remove all unnecessary dependencies, as described by Algorithm 12.

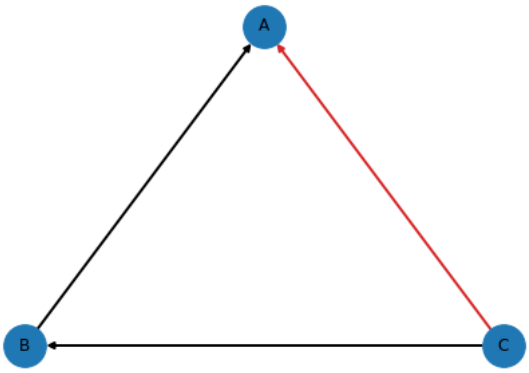


Figure 6: Example of an unnecessary dependency: the red edge can safely be removed

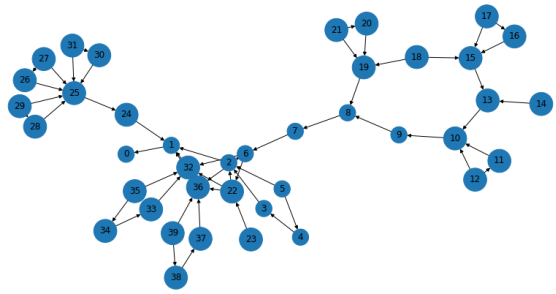
2. Compress all 'Linked Lists' in G. That means, detect all structures that are just a chain of nodes without extra dependencies, and replace them by one node with its frontier computed by Algorithm 5.

3. Detect all 'terminal' trees. That is, find all nodes v such that the subgraph of G rooted at v is a tree. As in step 1, compress all such trees into one node with the frontier computed by Algorithm 9.

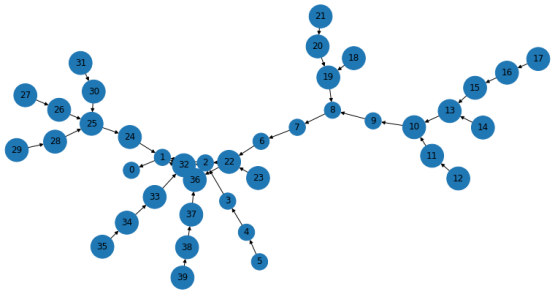
After these steps, we G might become a tree, in which case we can compute the frontier directly. If G is still not a tree, we can try approximating the Pareto Front of G by randomly sampling the frontiers of each node of G, and then applying Algorithm 3 to this new set of nodes.

Algorithm 13 BestEffort(G)

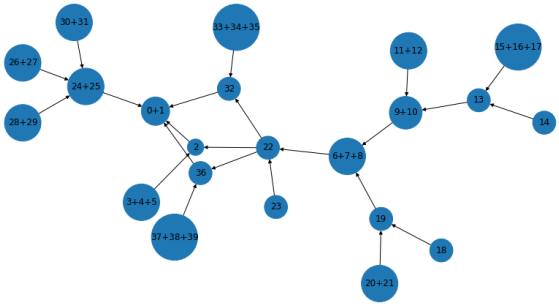
RemoveUnnecessaryDependencies(G)
Compress lists in G
if isTree(G) **then**
    return ParetoFrontTree(G)
end if
Compress trees in G
CurrentSize $\leftarrow \Pi_{v \in V(G)} |\mathcal{F}_v|$
ThresholdSize $\leftarrow$ ComputeThresholdSize(G)
while CurrentSize $>$ ThresholdSize **do**
    $v \leftarrow$ node with the largest frontier in G
    $\mathcal{F}_v \leftarrow$ Sample $\frac{1}{2}$ of $\mathcal{F}_v$
    CurrentSize $\leftarrow$ CurrenSize / 2
end while
return NaiveParetoFronDag(G)



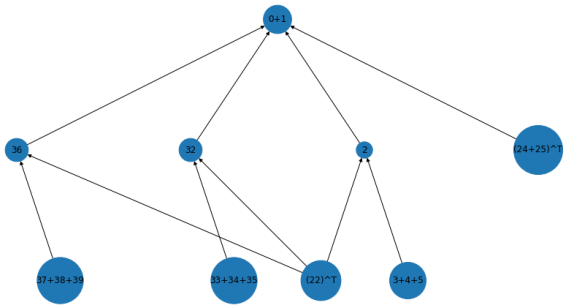
(a) Step 1: The original graph has many splits



(b) Step 2: Most splits were removed by RemoveUnnecessaryDependencies. However, node 22 is still a problem



(c) Step 3: All lists were compressed, they are marked by a '+' sign. For example, a list A -> B would become the node A+B



(d) Step 4: The subtrees under nodes 22 and 24 + 25 were compressed as well. The graph is now relatively small, but the frontiers of 22^T or $(24+25)^T$ may be huge. We reduce their sizes by sampling randomly, then apply NaiveParetoFrontDag.

Figure 7: Visual representation of how BestEffort computes the Pareto Frontier of a particular DAG

Algorithm 13 offers a rigorous description of how we approximate Pareto Frontier of any general DAG structures. The algorithm makes use of a certain function ComputeThresholdSize, that we did not define anywhere. Indeed, it is up to the user to define this function based on how they want the approximation to behave. We leave here some general remarks about the function:

- The bigger the threshold, the higher the quality of the approximation, but the longer the running time of $BestEffort$
- In general, different graphs can compute a variable number of configurations per second. By handling a configuration we mean evaluating Algorithm 1 and Algorithm 2. From empirical experiments we concluded that (on the same hardware) the number of configurations is solely dependent on the size of the graph (as seen in Figure 13 in Appendix A).

6 Results

6.1 Methodology

To measure the quality of our algorithm, we used the TPC-H ([9]) and TPC-DS ([8]) benchmarks.

```
def gen_random_frontier(k_points=5):
    TYPES = (
        (2, 60),
        (7, 70),
        (10, 20),
        (5, 15),
        (15, 50),
        (12, 84),
        (48, 10)
    )
```

```
    def _gen_pareto_points(n, latency, coeff):
        # cost * latency ~= coeff
        latencies = np.random.uniform(latency / 2, 2 * latency, n)
        costs = coeff / latencies
        confs = [Point(latency=lat, cost=c) for (lat, c) in zip(latencies, costs)]
        return confs

    stage_type = np.random.choice(np.arange(0, len(TYPES)))
    latency, coeff = TYPES[stage_type]
    frontier = _gen_pareto_points(k_points, latency, coeff)
    return frontier
```

Our methodology consisted of the following steps:

1. Get the stage dependencies of the graphs in the TPC-H and TPC-DS benchmarks. Store them in networkx DiGraph format.
2. For each node of each graph, generate a random Pareto Frontier using the python function pasted above. To avoid getting too favorable / unfavorable scenarios for specific graphs, we randomly generated 3 independent versions of each frontier.
3. Run Algorithm 3 on each graph. For some graphs, this step took more than 1 day to finish.
4. Run Algorithm 13 on each graph. For ComputeThresholdSize, for every different graph size n among our graph pool, we computed the average number of configurations per second at this size. The ThresholdSize was then a function of n that gave our naive algorithm t seconds to complete. We used 20 different values of t, equally spaced between 0.1 and 2 seconds.

It should be noted that the frontier sizes of stages were chosen so that the naive algorithm can complete in reasonable time (i.e no more than 2 days on our hardware). This resulted in some big graphs having nodes with frontier size of just 2 or sometimes even 1. While this is not consistent with production cases, where most nodes have 5 or more points in their frontier, it was unfortunately not possible to do better due to the time constraints of submitting this report.

6.2 Measuring the quality of Pareto Frontiers

To asses the quality of a Pareto Frontier, we use a standard metric of Multi-Objective Optimization: the Hypervolume Indicator. However, in our problem, we are dealing with different objective spaces (in the sense that the cost and latency ranges of different graphs). To adjust for this, we will normalize by the Hypervolume of the reachable objective space of our graph. We borrow the definition of the Utopia point from [7] and define the reference point as in Algorithm 14. If we denote r to be the reference point of a graph G and u its utopia point, our quality measure of a certain frontier $\mathcal{F}$ will be computed as:

$$\text{AdjutedHypervolume}(\mathcal{F}) = \frac{\text{HypervolumeIndicator}(\mathcal{F}, r)}{(r_2 - u_2) * (r_1 - u_1)}$$

Algorithm 14 GetReferencePoints(G)

$r \leftarrow$ root(G)
MaxCost $\leftarrow \sum_{v \in V} \sup p \in \mathcal{F}_v p_2$
MaxLatency $\leftarrow$ MaxLatencyAt(r)
RefPoint $\leftarrow$ (MaxLatency, MaxCost)
MinCost $\leftarrow \sum_{v \in V} \inf p \in \mathcal{F}_v p_2$
MinLatency $\leftarrow$ MinLatencyAt(r)
Utopia $\leftarrow$ (MinLatency, MinCost)
return Utopia, RefPoint

Algorithm 15 FrontiersEqual($\mathcal{F}_1$, $\mathcal{F}_2$, ϵ)

for $p_1 \in \mathcal{F}_1$ **do**
  $d = \inf_{p_2 \in \mathcal{F}_2} \|p_1 - p_2\|$
  if $d \geq \epsilon$ **then**
    p_1 is not in $\mathcal{F}_2$, frontiers are not equal
    return False
  end if
end for
for $p_2 \in \mathcal{F}_2$ **do**
  $d = \inf_{p_1 \in \mathcal{F}_1} \|p_1 - p_2\|$
  if $d \geq \epsilon$ **then**
    p_2 is not in $\mathcal{F}_1$, frontiers are not equal
    return False
  end if
end for
return True

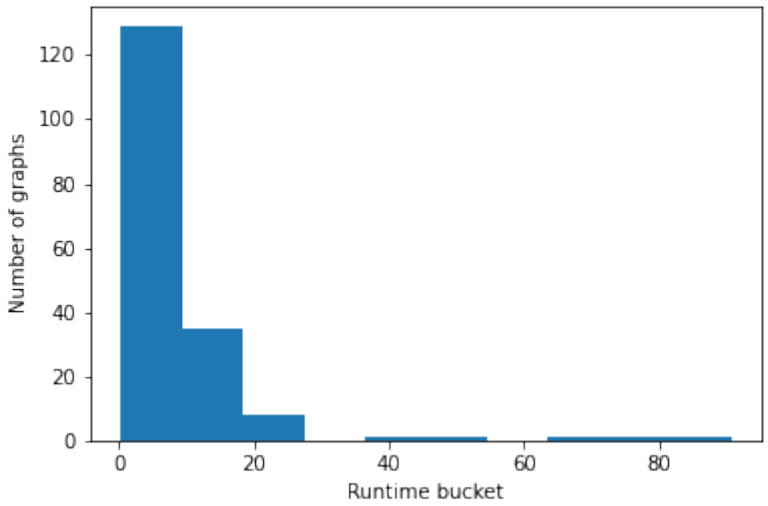


Figure 8: Histogram of $BestEffort$ on Trees

6.3 Overview of results

Throughout this section, we will say that two frontiers are equal if and only if Algorithm 15 evaluates to True on the two frontiers with $\epsilon = 10^{-4}$.

6.3.1 Trees and tree equivalents

In this section, we will study the results obtained for trees, as well for graphs that can be converted to a tree by calling Algorithm 12. We are interested in these results, as we know from subsection 4.4 that we can compute the full Pareto Front of any tree efficiently and from section 7 that this Frontier will always be the full Pareto Frontier of the given graph. In total, we have 213 graphs, from which 50 are trees, and 127 are tree equivalents. All in all, our efficient algorithms cover about 83% of all graphs. Note that the number of tree equivalents is subject to randomness, as it depends on the Pareto Frontiers of individual nodes.
As expected, all tree and tree equivalents have equal frontiers. Moreover, all trees computed extremely fast (less than 100ms), as seen in Figure 8.

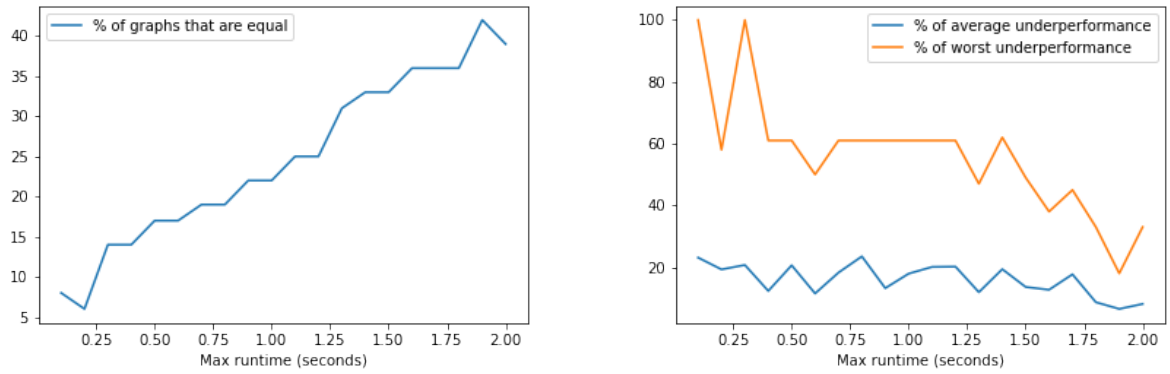


Figure 9: Main results summarized

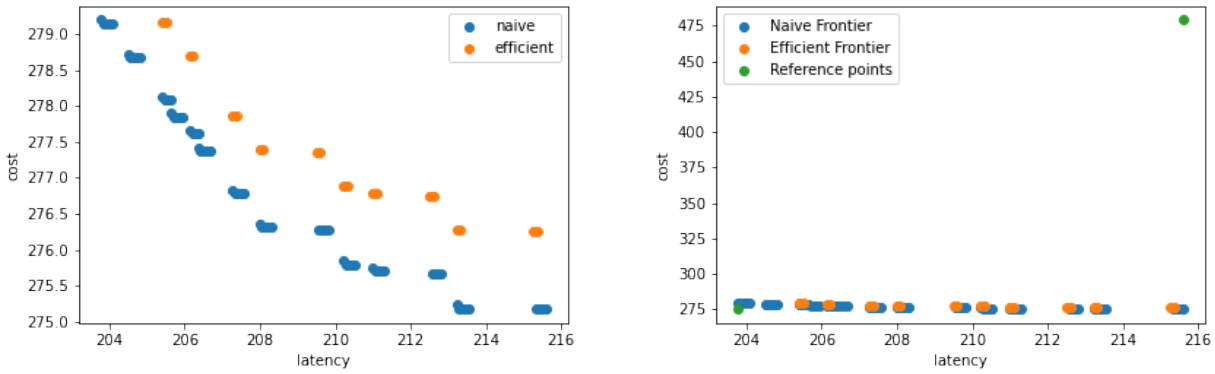
6.3.2 Other DAGs

In this section, we will be referring to *naive frontiers* as the frontiers of graphs computed by Algorithm 3, and *efficient frontiers* as the frontiers of graphs computed by Algorithm 13.
We use three metrics to assess the quality of our approximations:

1. The percentage of naive frontiers that are *equal* to efficient frontiers (i.e the number of graphs for which the naive and best effort algorithm had equal outputs).
2. The worst underperformance of the efficient algorithm. For a specific graph G, we measure the underperformance of the efficient algorithm as
$$\frac{\text{dominated space of } BestEffort(G)}{\text{dominated space of } NaivePareto(G)}$$
3. The mean underperformance of the efficient algorithm.

Figure 9 shows the three metrics we defined for different target running times we gave to the $BestEffort$ algorithm. As expected, the more time we give to $BestEffort$ for approximating, the better the approximations it yields. Indeed, going from 0.1 seconds maximum running time to 2 seconds maximum running time, the percentage of equal frontiers rose from below 10% to just above 40%. The worst and mean underperformance are also decreasing as we give more time to the approximation algorithm.

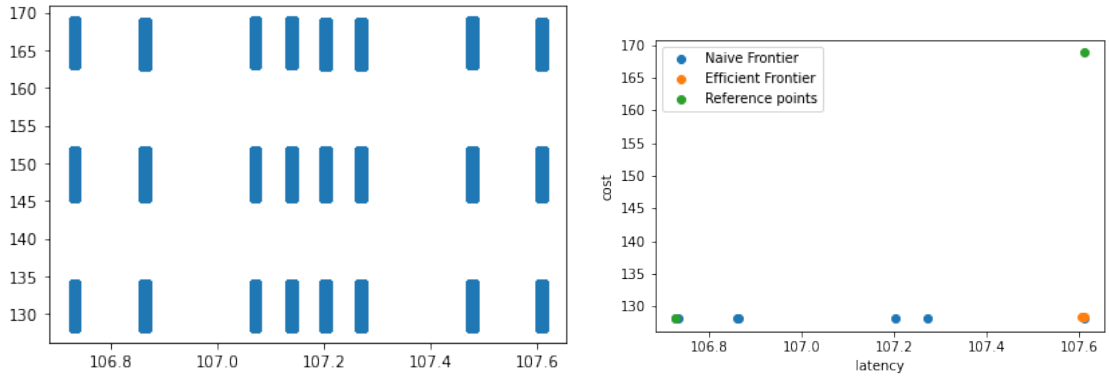
However, there are some clear fluctuations in the graph that we will study further. For most parameters t, the worst underperformance happens on the graph that we called TPC-DS-158 (see Figure 12). Clearly, our compression techniques were unable to significantly reduce the number of nodes, which means that the down-sampling of the frontiers of nodes that is done by $BestEffort$ will be significant, and we are likely to lose a lot of meaningful points.



(a) Pareto Front without reference points

(b) Pareto Front with reference points

Figure 11: TPC-DS-26

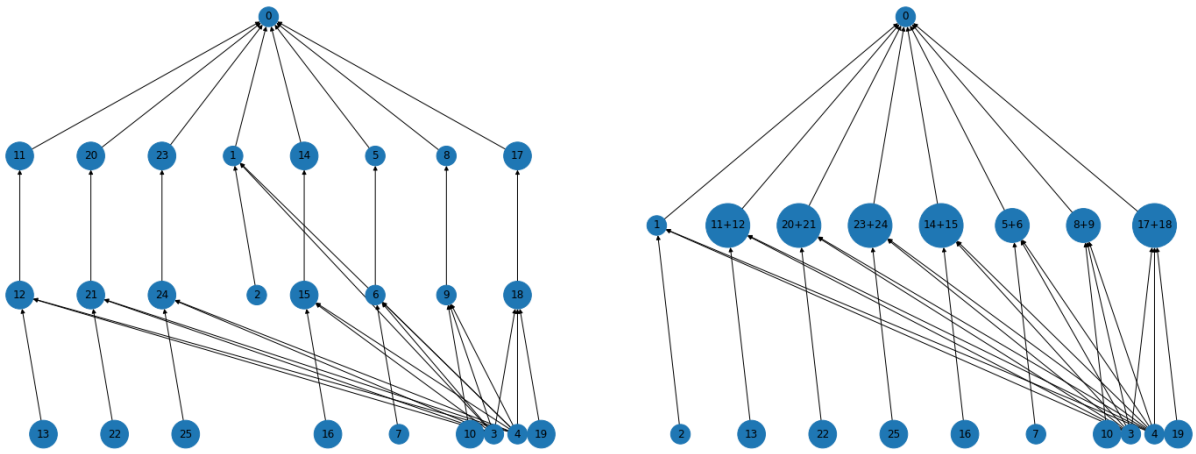


(a) All possible configurations

(b) Pareto Frontiers

Figure 10: TPC-DS-158 graph analysis

On top of that, by virtue of randomness, the size of the frontier of TPC-DS-158 is actually very small. In this case, only 8 points are found to be Pareto Optimal. Indeed, while TPC-DS-158 has about 16 million possible configurations, all of them end up if one of 8 straight lines, as seen in Figure 10a. Due to the down-sampling, most of these points are lost, which the dominated space significantly. This can be seen in Figure 10b.
We would like to note that, although it is important to take such cases into account, it is also important to acknowledge that they occur quite rarely. As such, only TPC-DS-158 had this problem due to its unusual configurations.
A more ordinary example of underperformance of the $BestEffort$ algorithm can be seen on the example of TPC-DS-26 (Figure 11). This example is much more normal in the sense that our approximation is just a 'shifted' version of the true frontier, with slightly less points. When the scale is adjusted for reference points, we see that the distances between the true frontier and the approximation are actually quite small.



(a) Original graph

(b) After applying various transformations from $BestEffort$

Figure 12: TPC-DS-158 graph

7 Proofs And Complexity Analysis

7.1 A small note on how to read this section

In this section, we will prove the correctness of Algorithm 5 and Algorithm 6. For this, we need will use two 'utility' lemmas: Claim 1 and Claim 2. The goal of these lemmas is simply to prove the correctness of the 'merge' step of the Divide-And-Conquer approach we used in our algorithms. In other words, we want to prove that the Pareto Front of a list can be obtained by the following sequence of operations:

1. Split a list in two sub-lists. For example, in Figure 2 that could be [A, B] and [C, D].
2. Compute the Pareto Front of each part separately.
3. Merge the Pareto Fronts of the two parts. Here we mean perform pairwise addition on the two sets, and only keep the Pareto optimal points from the result.

Since the proofs are very similar for both lists and joins, we also introduce a general addition operation (which can just be treated as coordinate-wise addition in $\mathbb{R}^d$). This can be useful for a potential generalization to many accumulators.
Admittedly, the proofs of these lemmas are convoluted and may be hard to follow. With that in mind, we suggest the reader starts reading this section with Theorem 1.
All other proofs apart from the claims mentioned above can be read in any order.

7.2 Definitions

We borrow the definition of a configuration from subsection 2.3, i.e we define a configuration on $G = (V, E)$ as a mapping

$$\omega : v \in V \mapsto \omega(v) \in \mathcal{F}_v \subseteq \mathbb{R}^d$$

that maps every node of G to one of it's available configurations. We define $\Omega(G)$ to be set of all possible configurations on G. We also define the evaluation mapping, that maps a configuration $\omega \in \Omega(G)$ to the result of running G under ω, i.e

$$E_G : \omega \in \Omega(G) \mapsto F_G^\omega \in \mathbb{R}^d$$

Under this, we can define the objective space of G

$$\Phi_G = \{E_G(\omega),\ \omega \in \Omega(G)\}$$

We also define the function Pf, which takes a set of points in $\mathbb{R}^d$ and outputs its Pareto front. For example, the Pareto front of a graph G is $\mathcal{F}_G = Pf(\Phi_G)$.
Throughout the section we also use the definition of 'Pareto domination', which we will borrow from subsection 2.4.

7.3 Claims

Claim 1. *Let*

$$+ : (F_1, F_2) \in \mathbb{R}^d \times \mathbb{R}^d \mapsto F_1 + F_2 \in \mathbb{R}^d$$

be a commutative and associative operation such that, for every graph
$G = (V, E)$ *that is either a 'Linked List' or a list of children in a 'Join', the evaluation function can be expressed as*

$$E_G(\omega) \mapsto \sum_{v \in V} \omega(v)$$

(note that in the sum we use the addition operation from above). Let $A = (V, E)$ *be some graph that is either a 'Linked List' or a 'Join' and let* $H, G \subseteq V$ *be two sets of vertices such that* $H \cup G = A$*, and* $H \cap G = \emptyset$*. Then*

$$Pf(\mathcal{F}_H \oplus \mathcal{F}_G) \subseteq \mathcal{F}_A$$

Where by $\oplus$ *we denote Minkowski sum of sets*

$$X \oplus Y = \{x + y, \forall (x, y) \in X \times Y\}$$

Proof. Let $F \in Pf(\mathcal{F}_H \oplus \mathcal{F}_G)$. By definition of the Minkowski sum, we know that there exists a pair of points $F_H \in \mathcal{F}_H$, $F_G \in \mathcal{F}_G$ such that

$$F_H + F_G = F.$$

Finally, choose $\omega_H \in E_H^{-1}(F_H)$ and $\omega_G \in E_G^{-1}(F_G)$ two configurations on H and G respectively that lead to points F_H, F_G.

We will prove by contradiction that $p \in \mathcal{F}_A$.
Assume $p \notin \mathcal{F}_A$. Define

$$\omega_A : v \mapsto \begin{cases} \omega_H(v) & \text{if } v \in H \\ \omega_G(v) & \text{if } v \in G \end{cases}$$

, we know that ω_A is a valid configuration on A since $H \cup G = A$, and $H \cap G = \emptyset$. Since F is just the sum F_H and F_G, we also deduce that $F = E_G(\omega_A) \in \Phi_A$.
Hence $F \in \Phi_A \setminus \mathcal{F}_A$ (so F is an objective that is not Pareto optimal), therefore there must exist a point $F' \in \mathcal{F}_A$ that Pareto dominates F. We can decompose the configuration $\omega' \in E_A^{-1}(F')$ into two configurations $\omega'_H = \omega'|_H$ and $\omega'_G = \omega'|_G$ on H and G respectively. This also yields two points $F'_H = E_H(\omega'_H) \in \mathcal{F}_H$, $F'_G = E_G(\omega'_G) \in \mathcal{F}_G$ such that

$$F'_H + F'_G = F'$$

We get that $F'_H + F'_G$ dominates $F_H + F_G$.
However, $F'_H + F'_G \in \mathcal{F}_H \oplus \mathcal{F}_G$ and $F_H + F_G \in Pf(\mathcal{F}_H \oplus \mathcal{F}_G)$. By definition of the Pf function, $F = F_H + F_G$ cannot be dominated by $F' = F'_H + F'_G \in \mathcal{F}_H \oplus \mathcal{F}_G$, which contradicts the definition of F' and concludes the proof. □

Claim 2. *Let*

$$+ : (F_1, F_2) \in \mathbb{R}^d \times \mathbb{R}^d \mapsto F_1 + F_2 \in \mathbb{R}^d$$

be a commutative and associative operation such that, for every graph
$G = (V, E)$ *that is either a 'Linked List' or a 'Join', the evaluation function can be expressed as*

$$E_G(\omega) \mapsto \sum_{v \in V} \omega(v)$$

(note that in the sum we use the addition operation from above). Let $A = (V, E)$ *be some graph that is either a 'Linked List' or a 'Join' and let* $H, G \subseteq V$ *be two sets of vertices such that* $H \cup G = A$*, and* $H \cap G = \emptyset$*. Then*

$$\mathcal{F}_A \subseteq Pf(\mathcal{F}_H \oplus \mathcal{F}_G)$$

Where by $\oplus$ *we denote Minkowski sum of sets*

$$X \oplus Y = \{x + y, \forall (x, y) \in X \times Y\}$$

Proof. Let $F \in \mathcal{F}_A$ and let $\omega \in E_A^{-1}(F)$ be some configuration that leads to F. Let $\omega_H = \omega|_H$ and $\omega_G = \omega|_G$ be the projections of ω on H and G. Finally, set $F_H = E_H(\omega_H)$ and $F_G = E_H(\omega_H)$. By definition, we know that

$$F = F_H + F_G$$
$$F_H + F_G \in \mathcal{F}_H \oplus \mathcal{F}_G$$

and thus $F \in \mathcal{F}_H \oplus \mathcal{F}_G$.
We now will prove by contradiction that $F \in Pf(\mathcal{F}_H \oplus \mathcal{F}_G)$.

Indeed, assume $F \in (\mathcal{F}_H \oplus \mathcal{F}_G) \setminus Pf(\mathcal{F}_H \oplus \mathcal{F}_G)$. Then there exists a point $F' \in Pf(\mathcal{F}_H \oplus \mathcal{F}_G)$ that dominates F. However, by Claim 1, we know that $F' \in \mathcal{F}_A$. We have

$$F \in \mathcal{F}_A$$
$$F' \in \mathcal{F}_A$$
$$F' \text{ dominates } F$$

This is impossible, and hence the initial assumption is False. This concludes the proof. □

Theorem 1. *Algorithm 5 always output the full Pareto front of the input 'Linked List'*

Proof. We will prove this by induction on the size of the list n. First, assume $n = 1$. This case is trivial as each node stores the Pareto front of the underlying stage.
Now assume $n \geq 1$ and we want to compute the Pareto front of a list of size $n + 1$. First, we split the list in two lists of size at most $\lceil \frac{n+1}{2} \rceil \leq n$. Thus, by the induction hypothesis, both $\mathcal{F}_H$ and $\mathcal{F}_G$ (as defined in the algorithm) will contain the full pareto fronts of the 'upper' and 'lower' parts of $\mathcal{S}$, respectively. Observe that the return value of Algorithm 5 is equivalent in our notation to $Pf(\mathcal{F}_H + \mathcal{F}_G)$, where $+$ is defined on $\mathbb{R}^2 \times \mathbb{R}^2$ as

$$\begin{pmatrix} latency(p_1) \\ cost(p_1) \end{pmatrix} + \begin{pmatrix} latency(p_2) \\ cost(p_2) \end{pmatrix} = \begin{pmatrix} latency(p_1) + latency(p_2) \\ cost(p_1) + cost(p_2) \end{pmatrix}.$$

We now use Claim 1 and Claim 2 (noting that $+$ is trivially associative and commutative). From there we conclude that

$$Pf(\mathcal{F}_H \oplus \mathcal{F}_G) = \mathcal{F}_A \qquad (*).$$

Observe that the LHS of $(*)$ is the ouput of Algorithm 5, and the RHS is the full Pareto front of A. Hence $(*)$ concludes the proof. □

Theorem 2. *Algorithm 7 always output the full Pareto front of the input 'Join'*

Proof. We proceed exactly as in Theorem 1, but we use for $+$ the operation

$$\begin{pmatrix} latency(p_1) \\ cost(p_1) \end{pmatrix} + \begin{pmatrix} latency(p_2) \\ cost(p_2) \end{pmatrix} = \begin{pmatrix} \max(latency(p_1), latency(p_2)) \\ cost(p_1) + cost(p_2) \end{pmatrix}.$$

Which is again trivially associative and commutative. □

Claim 3. *Let $\mathcal{A}$ be an array with N nodes and define*

$$M := \max_{\mathcal{F} \text{subarray of } \mathcal{A}} |Pf(\mathcal{F})|.$$

The complexity of Algorithm 5 is at most

$$O(M^2 logN)$$

Proof. Observe that, in Algorithm 5 the size of the input is divided by two at every interation. At every iteration, we use Algorithm 4 to merge two subarrays $\mathcal{L}$ and $\mathcal{R}$. By assumption, we know that

$$|Pf(\mathcal{L})| \leq M, |Pf(\mathcal{R})| \leq M,$$

and hence the call of Algorithm 4 would require at most $O(M^2)$ operations. We thus conclude that the overall complexity is thus $O(M^2 logN)$, as required. □

Claim 4. *Let $\mathcal{A}$ be an array with N nodes and define*

$$M := \max_{\mathcal{F} subarray\ of\ \mathcal{A}} |Pf(\mathcal{F})|.$$

The complexity of Algorithm 7 is at most

$$O(M^2 logN)$$

Proof. Again, the proof is the exact same as in Claim 3 since the only difference between the algorithms is the 'merge' step of Divide And Conquer, which has in this case the same complexity $O(M^2)$ □

Claim 5. *Every tree $G = (V, E)$ can be decomposed into a collection inter-connected 'Joins' and 'Linked Lists'*

Proof. The proof is outside the scope of the paper. We will refer to Algorithm 9 as a 'proof'. □

Theorem 3. *Let $G = (V, E)$ be a tree. Algorithm 9 will return the full Pareto Front of G.*

Proof. We will prove by induction on the combined number n of different 'Joins' and 'Linked Lists' of G. Let us start with the base case of G being a 'Linked List' or a 'Join' structure graph. In those cases, we have $n = 1$, and the correctness is guaranteed by Theorem 1 and Theorem 2 respectively.
Assume now that the hypothesis is true for some $n \in \mathbb{N}$, and let G be a graph with $n + 1$ 'Linked Lists' or 'Joins'. We now separate G into two subgraphs:

1. The 'root structure' of G, which is either the 'Join' formed by the root of G and it's direct children, or the maximal 'Linked List' from the root of G
2. G', a subgraph which contains everything but the root structure of G

With this splitting, we know that G' contains exactly n base structures, and hence we can apply Algorithm 9 to it. Knowing the Pareto fron of G', we can replace it by a single node $v_{G'}$ in G, in the sense defined by subsection 4.3. Now we can simply apply Theorem 1 and Theorem 2 depending on the root structure of G. Thus the claim holds for any $n \in \mathbb{N}$. By induction and Claim 5, we conclude the proof. □

Claim 6. *Let $G = (V, E)$ be a graph and $\mathcal{F}_G$ its Pareto Front.*
Let $G' = (V, E')$ be the result of RemoveUnnecessaryDependecies(G) and $\mathcal{F}_{G'}$ be its Pareto Front.
We have

$$\mathcal{F}_G = \mathcal{F}_{G'}$$

Proof. Let $F \in \mathcal{F}_G$, $\omega = E_G^{-1}(F)$, and $F' = E_{G'}(\omega)$. In other words, F and F' are the result of applying the same configuration ω to both G and G'. We will prove that $F = F'$. First, we know that

$$F_2 = F'_2 = \sum_{v \in V} \omega(v)_2$$

as defined by Algorithm 2, and hence the costs of running both graphs are the same.
As for the latency, we know that $E = E' \cup R$, where R is some set of removed edges. For every edge $(c, p) \in R$ (a tuple (u, v) is equivalent to an edge *from* u *to* v), we know that, for G

$$\text{LatencyAt}_G(p, \omega) = \omega(p)_1 + \max_{x \in \text{children}(p)} \text{LatencyAt}_G(x)$$

and for G'

$$\text{LatencyAt}_{G'}(p, \omega) = \omega(p)_1 + \max_{x \in \text{children}(p) - \{c\}} \text{LatencyAt}_{G'}(x)$$

However, by Algorithm 12, we know that there exists an $x \in \text{children}(p)$ such that MinLatencyAt(x) $\geq$ MaxLatencyAt(c). In particular, $\omega(x)_1 \geq \omega(c)_1$, and hence

$$\max_{x \in \text{children}(p)} \text{LatencyAt}_G(x) = \max_{x \in \text{children}(p) - \{c\}} \text{LatencyAt}_{G'}(x)$$

From this we conclude that the latency at p is equal for both G and G' under ω. Since this is true for any edge $(c, p) \in R$, and G and G' share all other edges, we know that the evaluation of LatencyAt are the same for every node, and, in particular, at the root r of G (and G'). We thus conclude

$$F_1 = \text{LatencyAt}_G(r) = \text{LatencyAt}_{G'}(r) = F'_1$$

and

$$F = F'.$$

Since $F = F' \in \mathcal{F}_{G'}$, we know that $F \in \mathcal{F}_{G'}$. Since F was chosen at random in $\mathcal{F}_G$, we finally write

$$\mathcal{F}_G \subseteq \mathcal{F}_{G'}.$$

The proof is symmetrical for the inclusion $\mathcal{F}_{G'} \subseteq \mathcal{F}_G$. □

8 Future Work

There are many things that could be done to improve this project. First, we will note that the assumptions made at the beginning of this paper do not allow for applying our algorithm to real-world problems in their current form. In particular, Assumption 1 is unrealistic, as in actual Spark queries, there are parameters that are shared among all nodes. We hope that in our future work we manage to drop this assumption, while making minimal changes to the algorithms presented here.
Furthermore, the approximations made by Algorithm 13 could be improved if needed, by improving the efficiency of Algorithm 3. One of the ways this can be done is by discarding some configurations immediately. For example, in the case of 'Join' structures where one child is long running, we can immediately determine that the optimal configuration for other (short-running) children are ones with the highest latency and lowest cost, as this does not impact the latency of running the 'Join', but reduces its cost.
Another way to limit the time of iterating through all configurations is only iterate through the subgraphs rooted at nodes with multiple children. We can do this by proceeding as follows:

1. Compute the frontier of the subgraph rooted at a problematic split s.
2. In the original graph, replace this subgraph a node with just one frontier point
3. Compute the frontier of the graph as if this was a tree, but make sure to avoid counting the cost of s multiple times
4. Do this for every point in the frontier of s

With this approach, our complexity is not exponential, but limited to the complexity of computing the tree multiplied by the number of nodes in the frontier of the subgraph rooted at s.
Finally, one could also consider a fundamentally different approach to approximation, where for all nodes with multiple parents we choose one edge uniformly at random and only keep this edge.

9 Conclusions

In this thesis, we presented an efficient algorithm to compute the Pareto Frontier of computational graphs with multiple optimization objectives. The main finding of this paper is an algorithm that is both efficient and proven to be correct for any tree. We then present a heuristic to convert some non-tree graphs into tree, while preserving the Pareto Front. With this heuristic, our efficient algorithm covers about 83% of production test cases, which were taken from well-known benchmarks for Spark. For the remaining 17%, we give an approximation algorithm, which very often provides high-quality results, while still yielding a speedup of thousands of times compared to the naive algorithm. We finish with a short paragraph providing guidance on how to improve our work in this project.

We believe that the techniques presented in this paper will enable big leaps forward for offering the best cost-performance benefits to real-world users and cloud providers, ultimately leading to long-term sustainable computing.

10 References

[1] Amazon ec2 instance types. https://aws.amazon.com/ec2/instance-types/, 2019.

[2] Amazon aurora serverless. https://aws.amazon.com/rds/aurora/serverless/.

[3] Python implementation of the algorithms presented in this paper. https://github.com/Philippe-Guyard/Bachelor-Thesis.

[4] Chenghao Lyu, Qi Fan, Fei Song, Arnab Sinha, Yanlei Diao, Wei Chen, Li Ma, Yihui Feng, Yaliang Li, Kai Zeng, et al. Fine-grained modeling and optimization for intelligent resource management in big data processing. *arXiv preprint arXiv:2207.02026*, 2022.

[5] Oracle cloud infrastructure data flow. https://www.oracle.com/big-data/data-flow/, 2020.

[6] Kaushik Rajan, Dharmesh Kakadia, Carlo Curino, and Subru Krishnan. Perforator: eloquent performance models for resource optimization. In *Proceedings of the Seventh ACM Symposium on Cloud Computing, Santa Clara, CA, USA, October 5-7, 2016*, pages 415–427, 2016.

[7] Fei Song, Khaled Zaouk, Chenghao Lyu, Arnab Sinha, Qi Fan, Yanlei Diao, and Prashant Shenoy. Spark-based cloud data analytics using multi-objective optimization. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*, pages 396–407. IEEE, 2021.

[8] Tpc-ds benchmark. https://www.tpc.org/tpcds/.

[9] Tpc-h benchmark. https://www.tpc.org/tpch/.

A Appendix

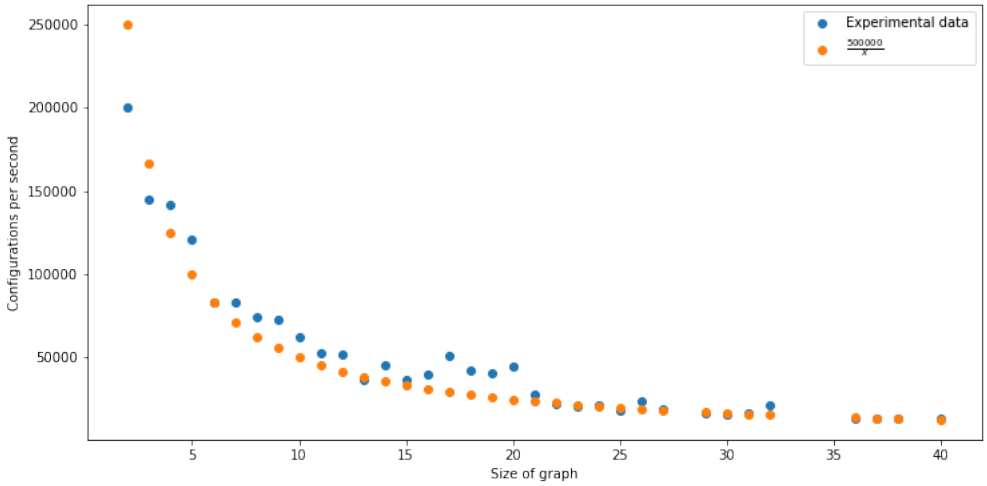


Figure 13: Graph size versus number of configurations per second that can be handled

We also published the code that was written to accompany this thesis on Github. It can be found by following the reference [3]

Statement of Academic Integrity Regarding Plagiarism

I, the undersigned Philippe Guyard [family name, given name(s)], hereby certify on my honor that:

1. The results presented in this report are the product of my own work.

2. I am the original creator of this report.

3. I have not used sources or results from third parties without clearly stating thus and referencing them according to the recommended rules for providing bibliographic information.

Declaration to be copied below:

I hereby declare that this work contains no plagiarized material.

Date March, 18th 2023

Signature