DAG*: A Novel A*-alike Algorithm for Optimal Workflow Execution across IoT Platforms

Errikos Streviniotis
School of ECE
Technical University of Crete
estreviniotis@tuc.gr

Dimitrios Banelas
School of ECE
Technical University of Crete
dbanelas@tuc.gr

Nikos Giatrakos
School of ECE
Technical University of Crete
ngiatrakos@softnet.tuc.gr

Antonios Deligiannakis
School of ECE
Technical University of Crete
adeli@softnet.tuc.gr

***Abstract*—Many IoT applications from diverse domains rely on real-time, online analytics workflow execution to timely support decision making procedures. The efficient execution of analytics workflows requires the utilization of the processing power available across the cloud to edge continuum. Nonetheless, suggesting the optimal workflow execution over a large network of heterogeneous devices is a challenging task. The increased IoT network size increases the complexity of the optimization problem at hand. The ingested data streams exhibit highly volatile properties. The population of network devices dynamically changes. We introduce DAG*, an A*-alike algorithm that prunes large amounts of the search space explored for suggesting the most efficient workflow execution with formal optimality guarantees. We provide an incremental version of DAG* retaining the optimality property. Our experimentation in real-world scenarios shows that DAG* suggests the optimal workflow execution with 3 to 31 orders of magnitude fewer iterations compared to the entire search space size, outperforming heuristics employed in prior state of the art up to x4.5 wrt the goodness of the suggested workflow.**

***Index Terms*—IoT, Optimization, Data Streams, A* Algorithm**

I. Introduction

Many diverse applications from the smart city, smart grid [1], [2], smart factory [3], and various other domains operate over geo-distributed infrastructures, composed of powerful clouds, intermediate fog and edge devices. In such Internet-of-Things (IoT) settings, devices continuously monitor their operational realm and produce unbounded streams of data at a high rate [4]. These data need to be processed in an online, real-time fashion in the scope of analytics workflows so as to continuously deliver useful output supporting reactive and proactive decision making procedures [5], [6]. IoT networks use hierarchical organizations [2], [7] to reduce latency, enhance scalability, and manage bandwidth efficiently. Optimized data flow and resource allocation stems from delegating tasks across the edge, fog, and cloud layers. This allows lower layers (edge and fog) to handle preliminary data processing, filtering, and aggregation close to the devices, reducing raw data volume sent to the cloud and lowering transmission costs. By processing data closer to the source, the network minimizes cloud dependencies, conserves resources and improves response times for real-time applications.

This work was supported by the CREXDATA project, which received funding from the European Union's Horizon Europe Programme, under grant agreement No 101092749.

Consider a smart city infrastructure and an analytics workflow, as the one in Figure 1, which performs higher order aggregation and stores generated plots [1]. Such analytics workflows are composed of various operators for machine learning, stream transformation or user defined functions, each processing data ingested from various devices [1]. Having IoT devices naively relay sensed data to the cloud side is severely suboptimal since the processing capacity of the devices is never exploited. To fully exploit the processing power of the entire cloud-to-edge continuum, some of the operators of the workflow can be assigned for execution on the fog or edge devices so that, aggregative results can be extracted early, and only more compact data representations will be delivered to the cloud side. In that, network and processing latencies can be diminished [8], [9]. In turn, reduced latency is important for the real-time, continuous delivery of the analytics outcomes. Nonetheless, the problem of optimal operator assignment to network devices, represented as disks in Figure 1, is NP-hard [10], [11]. In Figure 1 we can see that each operator may be assigned for execution to different devices (disks), while the possible combinations for all operators of the workflow, is difficult to track. In IoT settings, optimizing the execution of a workflow introduces further challenges related to the volatility of ingested stream statistical properties (such as rapidly changing skews, frequencies), the increased network size, which increases the complexity of the optimization problem, and the fact that new devices may enter or depart [11] at any time.

The problem of efficient analytics workflow execution has been extensively studied under several setups in prior research. From traditional optimization in centralized streaming settings [12], [13], to networked architectures [10], [14]–[16], cross-platform optimization scenarios [17]–[21], elastic stream processing [22], [23] at the cloud, sensor networks [24], [25] and recently over IoT platforms [9], [11], [26]. Observing these approaches and the optimization problems they solve at a higher level of abstraction [5], the common theme is that there is an exhaustive search algorithm that explores the entire search space of possible workflow operator assignments and guarantees to solve the optimization problem at hand by outputting an *optimal graph* (highlighted in green disks and arrows in Figure 1), prescribing for every operator of the workflow the device(s) on which it should get executed, so that aggregate performance measures are optimized. Subsequently,

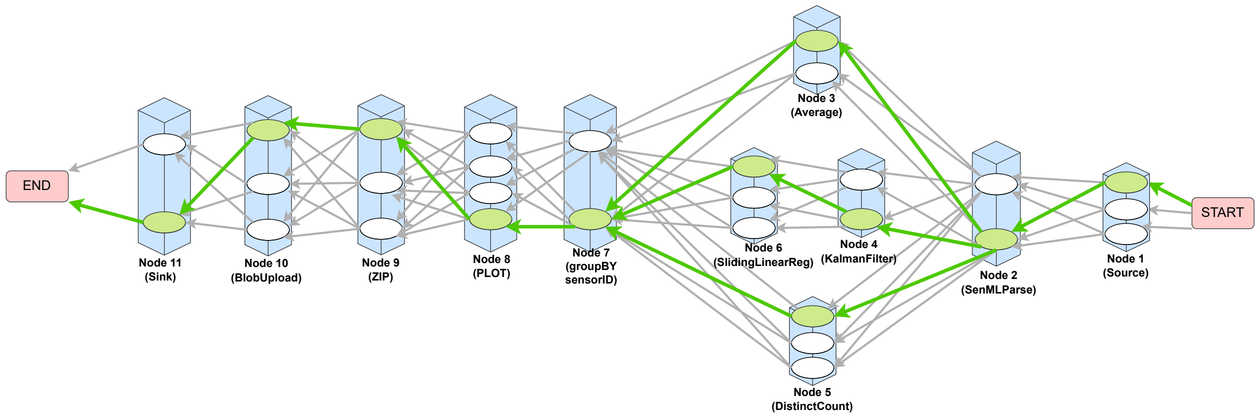


Fig. 1: Example of DAG* output on the STATS workflow [1]. Nodes are the operators of the analytics workflow. Disks in each node are possible configurations, i.e., executing the operator represented by Node i on network device d_k. Each disk has a different cost (processing latency). An edge denotes operator dependency with a different cost weight (network latency) depending on the configurations they connect. Green-colored edges and disks compose the optimal graph returned by DAG*.

each approach comes up with heuristics that reduce the portion of the explored search space, but end up with suboptimal (hopefully near-optimal) graphs describing the workflow execution suggestions. On the one hand, limiting the explored search space of the optimization algorithm is a desired feature so as to suggest an efficient workflow execution in a timely manner. On the other hand, relying on heuristics to provide more rapid suggestions may unpredictably compromise the goodness of the devised execution graph, since existing heuristics provide no a priori known quality guarantees [5].

In this work, we address this gap of all prior approaches, introducing an optimization algorithm which both substantially reduces the amount of search space it needs to explore, and simultaneously guarantees the optimality of the devised graph for the execution of the given workflow over the physical IoT infrastructure. Our algorithm is inspired by the A* path finding algorithm [27], [28]. A*'s power in efficient path finding stems from a couple of fundamental properties (Section VI): (a) the incorporation of a clever heuristic cost function, admissible only upon being an underestimation of the real cost of each path, during exploring the available search space; (b) the guarantee of optimality under the admissible heuristic precondition. The classic A* and any other path finding algorithm [28] are inappropriate in our setup because they would leave entire workflow operators out of the suggested workflow execution. For instance, the classic A* algorithm in a workflow graph like the one in Figure 1, will always output a path, i.e., only one out of Node 3, {Node 4, Node 6} or Node 5 would be included in the output solution. Our work is motivated by the intuition that if one could adapt and convey the aforementioned A* properties in the context of optimal graph finding, the novel optimal graph A*-alike algorithm would inherit the ability to prune large portions of the search space, also retaining optimality guarantees. Thus, the impact of the new algorithm for a variety of real scenarios explained above, would be high given the gap in prior art. In this paper we build a novel A*-alike algorithm for efficient optimal graph (workflow execution) discovery. Our contributions are:

- We propose DAG*, a novel A*-alike algorithm, for optimal workflow execution. DAG* is the first algorithm that guarantees the optimality of the suggested workflow execution graph, dramatically reducing the portion of search space being explored. To achieve that, in DAG*:
  - We redefine the concept of the admissible heuristic in the new context of optimal graph, instead of path discovery and we introduce a novel, admissible heuristic function.
  - We introduce two novel rules which guarantee that the new algorithm returns a graph instead of a path.
- We theoretically prove that DAG*, under these rules, always returns an optimal solution, i.e., the workflow execution graph that minimizes a user defined cost.
- We introduce d-DAG*, an incremental version of DAG* that boosts efficiency in dynamic IoT environments where cost and device population changes can occur unpredictably. Again, we provide theoretical optimality proof for d-DAG*.
- Our systematic experimental evaluation uses a well known, highly cited benchmark [1] from the smart city domain. Our results show that DAG* can provide optimal graphs, prescribing the execution of workflows over the physical IoT infrastructure, by exploring only a small fraction of the search space with 3 to 31 orders of magnitude fewer iterations compared to exhaustive search. We further show that DAG* outperforms state of the art heuristic paradigms [11], [26] by providing solutions with up to 4.5x better workflow execution performance. Our incremental algorithm, d-DAG*, further halves the amount of search space explored in volatile settings compared to running DAG* from scratch.
- DAG* has been incorporated in a commercial platform, namely the streaming extension of RapidMiner Studio [29] running as part of the Streaming Optimization service.

II. Problem Formulation

Consider a workflow as a *weighted Directed Acyclic Graph (DAG)* $\mathcal{G} = (V_{\mathcal{G}}, E_{\mathcal{G}})$, where $V_{\mathcal{G}}, E_{\mathcal{G}}$ are the sets of nodes and edges accordingly. Each node $i \in V_{\mathcal{G}}$ has a set of possible configurations denoted as C_i, where each configuration $c_k^i \in C_i$ is related with a cost $conf_c(c_k^i) \geq 0$. Note that $\forall i, j \in V_{\mathcal{G}}$, where $i \neq j$, the sets C_i and C_j may differ, while if a specific configuration c_k exists in both sets, then the corresponding cost of c_k may vary for each node, i.e., $conf_c(c_k^i) \neq conf_c(c_k^j)$. Moreover, each edge $e_{i\rightarrow j} \in E_{\mathcal{G}}$ has a non-negative weight that depends on the configurations that have been assigned to nodes i and j, denoted as $edge_c(c_k^i, c_l^j)$. On top of that, each node i has a set of predecessors $Pr(i)$, i.e., set of nodes that are connected with i via an edge $e_{j\rightarrow i}$, where $j \in Pr(i)$; and a set of successors $Su(i)$, i.e., set of nodes that are connected with i via an edge $e_{i\rightarrow j}$, where $j \in Su(i)$.

Also, given $\mathcal{G}$, we insert two *auxiliary nodes*, namely the START and the END nodes, to guide the initial and final steps of our algorithm (see Sections III-A and III-C). In particular, we connect the START node with each node $i \in V_{\mathcal{G}} : Pr(i) = \emptyset$ via an edge $e_{START\rightarrow i}$; while we connect the END node with each node $i \in V_{\mathcal{G}} : Su(i) = \emptyset$ via an edge $e_{i\rightarrow END}$. [1].

Illustrative Example: Consider the workflow graph of Figure 1. It includes interconnected nodes (workflow operators) that are executed to accomplish a higher order statistic computation analytic task. The interconnection between operators are the edges denoting the dependency of an operator on its previous ones. The workflow continuously ingests input data from Node 1 which is a `Source` operator, and each subsequent operator in the topologically sorted workflow continuously receives input from one or more other operators. Notably, a workflow can include both splits (the `SenMLParse` operator gives input to more than one other) and merges (the `groupBy sensorID` operator receives input from more than one operator). Each operator should be executed on some, among several network devices, though it has multiple device choices, termed configurations, with different costs/processing latencies ($conf_c$ in our problem formulation) represented as disks in Figure 1. Data can be transferred among the devices via $edge_c$ with a communication cost/latency equal to the weight of $edge_c$. The optimal graph of DAG* will have assigned each operator at a device aiming to minimize the entire cost (processing and communication latency) of the workflow as a whole.

DAG* finds the optimal graph based on some cost function. The solution that our algorithm will produce must contain $\forall i \in V_{\mathcal{G}}$, which should be assigned at a configuration $c_k^i \in C_i$.

Definition 1 (ANOC (AllNodesOneConfiguration) Graph)**.** *A graph $\mathcal{G}'$ is called ANOC if and only if the following three conditions hold: (a) $V_{\mathcal{G}'} = V_{\mathcal{G}}$; (b) one configuration is assigned to each node $i \in V_{\mathcal{G}'}$; and (c) all nodes $i \in V_{\mathcal{G}'}$ are chosen to be connected via a unique edge with each node $j \in Pr(i)$.*

Simply put, an ANOC graph is one that contains all nodes, one chosen configuration per node and one edge connecting chosen configurations of connected nodes. Notably, this definition is easily extensible to broader scenarios, where, for instance, a node can be replicated with multiple configurations. In that case, the said node just needs to be replicated in the graph $\mathcal{G}$.

Definition 2 (Optimization Problem)**.** *Given $\mathcal{G} = (V_{\mathcal{G}}, E_{\mathcal{G}})$, DAG* outputs the optimal graph, $\mathcal{G}^* = (V_{\mathcal{G}}^*, E_{\mathcal{G}}^*)$ such that:*

$$\textbf{\textit{Minimize:}}\ Real(\mathcal{G}^*) + \cancelto{0}{h(\mathcal{G}^*)} \tag{1}$$

$$\textbf{\textit{Subject to:}}\ \mathcal{G}^* \quad \textit{is an ANOC graph}$$

with the $Real(\mathcal{G}^*)$ and the heuristic $h(\mathcal{G}^*)$ costs explained in Equation 3 and Equation 2, later on.

We further point out that $h(\mathcal{G}^*)$ is the heuristic cost defined in Section III. As explained later in Equation 2, this heuristic cost is zero for the END operator of any given $\mathcal{G}$, i.e. just at the point when DAG* reaches the optimal solution for the entire workflow. We superfluously include $h(\mathcal{G}^*)$ (explicitly zeroing it out) in the above definition for completeness and for emphasizing the connection with Section III.

III. The Proposed DAG* Algorithm

A. Nodes' Heuristic Cost

DAG*, similar to the classic A* [27], utilizes a heuristic cost function for each node in $V_{\mathcal{G}}$ to guide its search. Remarkably, DAG* contributes with a novel formula for the heuristic cost of each node $i \in V_{\mathcal{G}}$, denoted as $h(i)$. This new formula is a prerequisite in order to operate aiming for optimal graphs instead of optimal paths. For a given workflow, the heuristic cost should be an optimistic estimation on what is the cost (aggregate network and processing latency) from a given node/operator (see Figure 1) to reach the END. For a heuristic to be admissible, i.e., guarantee optimality of DAG*, $h(i)$ should always be an underestimation of the real cost from a given node/operator to reach the END. The novel, admissible heuristic introduced by DAG* for optimal graph discovery is given by the following formula:

$$h(i) = \begin{cases} 0 \text{ if } i \in \{Pr(\text{END}), \text{END}\} \\ max_{j\in Su(i)}\{h(j) + min_{k\in C_j}\{conf_c(c_k^j)\}\} & \text{else} \end{cases} \tag{2}$$

We stress that the computation of the heuristics is performed in a reverse topological sort of the nodes; it starts with the END node,[2] proceeds with the $Pr(\text{END})$ set, then the nodes in $Pr(Pr(\text{END}))$ and so on. This process terminates by computing the heuristic of the START. Note also that Equation 2 leads to an *admissible* (heuristic) function since it never overestimates the real cost from any node $i \in V_{\mathcal{G}}$. This holds because, for each node i, Equation 2 computes the sum of the minimum configuration costs and the heuristics for each j, where $j \in Su(i)$. Then, in case of a node with multiple successors (e.g. Node 2 in Figure 1) we set the heuristic of i as the maximum computed sum of the minimums of its successors. This guarantees that our heuristic never overestimates the actual cost, since our approach attempts to discover the optimal

[1]All edges that begin from START and all edges that arrive at the END have a zero cost (or weight).

[2]Since the END node is a 'goal-node', its heuristic cost is equal to zero. Since it is a virtual node, it has a zero configuration cost as well.

graph, so all nodes are always contained in the solution. We highlight that we select the maximum of the minimums instead of the minimum of the minimums, which could also be used, since it offers a less (but still) optimistic and more realistic estimation of the actual cost. As such, the maximum of minimums gives an admissible heuristic simultaneously guiding our algorithm faster to the optimal solution. Now, for num_conf possible configurations per node, the $h(i)$ values can be precomputed using $O(|V_{\mathcal{G}}| \cdot (num\_conf + B))$ time, where B is the maximum number of successors $Su(i)$ of any node, by first computing the quantities $min_{k \in C_j}\{conf_c(c_k^j)\}$ in $O(num\_conf)$ time per node.

Illustrative Example: Consider that we want to compute the heuristic of Node 2, i.e., $h(2)$. We have that $Su(2) = \{\text{Node 3}, \text{Node 4}, \text{Node 5}\}$ with assumed heuristics $h(3) = 10$, $h(4) = 8$ and $h(5) = 12$. Moreover, for ease of exposition, assume that Node 3 and Node 5 have only one valid configuration with costs 5 and 1 accordingly, while Node 4 has two valid configurations with costs 5 and 7. Thus, from Equation 2 we get that $h(2) = max\{\{h(3) + 5\}, \{\{h(4) + min\{5, 7\}\}, \{h(5) + 1\}\} = max\{\{15\}, \{13\}, \{13\}\} = 15$.

DAG* also accounts for a real cost for each (partial or complete) graph of $\mathcal{G}$ that it has examined at any given point in time. This real cost, formally defined in the next section, for each operator of a given workflow, measures the cost from the START to the said node/operator. The real cost depends on (a) employed configurations for the visited nodes; (b) the weight of the edges that connect these configurations and (c) a chosen aggregation function.

B. Partial Graphs' Cost

Since our goal is to return the optimal graph, our algorithm extends partial graphs instead of just nodes. As we explain in Section III-C, DAG* maintains a priority queue where partial graphs are stored along with their corresponding costs. Each partial graph g contains the so far visited nodes and used edges of $\mathcal{G}$ and has some cost based on the configurations that have been assigned to its nodes. The *estimated cost* of a partial graph g is the sum of the real and the heuristic costs, formally:

$$f(g) = \underbrace{conf_c(c_k^i) + \underset{\substack{i \in T_g | c_k^i \\ j \in Pr(i) \in g^- \\ \ell \in C_j}}{aggr} \left\{Real(g^-) + edge_c(c_k^i, c_\ell^j)\right\}}_{Real(g)} + \underbrace{min_{i \in T_g}\{h(i)\}}_{h(g)} \tag{3}$$

where T_g is the set of the *terminal nodes* of partial graph g, i.e., those nodes with empty $Su()$ sets. The real cost $Real(g)$ is computed from the cost of g^-, which is a dequeued, visited partial graph. We add the cost of a new configuration for $i \in V_{\mathcal{G}}$ and we also add the weight of the edges that connect this configuration (disk in Figure 1) to the configurations of its predecessors in g^-. We `aggr`egate across all these edges using any function, e.g., sum, max, etc., depending on the application, i.e., serial execution may imply sum, while parallel execution may imply max.

Illustrative Example: Consider the scenario where $aggr = max$ and we just dequeued g^- which contains all Nodes 1 to 6 with a cost of $Real(g^-) = 56$. Now, we can extend g^- by inserting Node 7, which has a heuristic $h(7) = 8$, into one of its valid configurations. Assume that we generate g by selecting c_1^7 which has a cost of 4. Moreover, we assume that based on their applied configurations the communication costs from Nodes 3, 5, and 6 to 7 are equal to 3, 1, and 2 accordingly. Thus from Equation 3 we get that $f(g) = 4 + max\{\{56+3\}, \{56+1\}, \{56+2\}\} + 8 = 4 + 59 + 8 = 71$.

C. Extending a Partial Graph

DAG* uses an estimated cost-based priority queue to store partial graphs, during its quest of reaching the optimal graph it seeks for. Here, we describe the partial graph extension operation of our approach, as we visit more nodes towards optimal, complete graph discovery. To extend any partial graph g, g should be contained in a priority queue, denoted as Q, and should have the smallest estimated cost (Equation 3). Then g is selected and removed from Q. DAG* introduces two novel rules in order to ensure that the algorithm will output a graph instead of a path. The extension of partial graph g must follow these two rules:

Rule 1 During an extension, only one node i can be inserted to g along with the edges that lead to i and exist in $\mathcal{G}$, i.e., $\forall j \in Pr(i) : e_{j \to i} \in E_{\mathcal{G}}$.

Rule 2 A node i can be selected if and only if all nodes in $Pr(i)$ have been already inserted in g, i.e., $\forall j \in Pr(i) : j \in V_g$.

Rule 1 says that we must expand g selecting any, but one at a time, node i and Rule 2 says that, for a node to be qualified for selection, all its predecessors must have been included in g. Once we select i, we should generate $|C_i|$ (cardinality of C_i) new partial graphs that differ in the employed configuration of the newly inserted node i. Subsequently, we compute their estimated cost and we insert each of them into Q. For each expansion, the complexity for computing the costs of each of the $|C_i|$ new partial graphs is $O(|Pr(i)| \cdot num\_conf)$.

Lemma 1. *For any partial graph g that is stored in Q, it holds that $\forall i \in V_g$ all nodes in sets $Pr(i), Pr(Pr(i)), \cdots$ also exist in V_g, i.e., no node i can exist in any partial graph $g \in Q$ without all the nodes j that are connected with i, via an edge $e_{j \to i}$, in $\mathcal{G}$.*

Proof. We will prove this lemma by contradiction. Assume that at some iteration of DAG*, the priority queue Q encloses at least one partial graph g that contains node i and it holds that there exists at least one node j of i's predecessors, i.e., $\exists j \in Pr(i)$, such that $j \notin V_g$. Now, since DAG* never removes a node from a partial graph (once it has been added), this implies that at some point during the execution of DAG*, node i was added to a partial graph that did not contain node j. However, based on Rule 2 we know that DAG* can add node

Algorithm 1 DAG* algorithm for optimal graph discovery

Input: Graph $\mathcal{G}$, Configuration costs $conf_c$, Edge costs $edge_c$, Priority queue Q (default $Q = \emptyset$)
Output: Optimal graph $\mathcal{G}^*$

```
1: Insert the START and END nodes at $\mathcal{G}$
2: computeHeuristics($\mathcal{G}$, $conf_c$)
3: if $Q = \emptyset$ then
4:     $Q$.enqueue(START)
5: end if
6: $\mathcal{G}^* = \emptyset$
7: while $Q \neq \emptyset$ do
8:     $g = Q$.dequeue()
9:     if $g$ is ANOC then
10:        $\mathcal{G}^* = g$
11:        break
12:    end if
13:    $candidates$ = nodesMeetingRule2($\mathcal{G}, g$)
14:    $newNode$ = rand($candidates$)
15:    {Apply Rule 1}
16:    for all possible confs of $newNode$ do
17:        $tmpG$ = insertNode($newNode_i, \mathcal{G}, g, conf_c, edge_c$)
18:        computeCost($tmpG$)
19:        $Q$.enqueue($tmpG$, sortKey = estimated cost)
20:    end for
21: end while
22: return $\mathcal{G}^*$
```

i if and only if all nodes in $Pr(i)$ already exist in the partial graph. As a result, we have a *contradiction*. □

Illustrative Example: In the initial iteration, DAG* inserts to Q a partial graph g_0 that contains only the START node. g_0 is then dequeued and extended. In Figure 1, this extension process will enqueue 3 partial graphs, because Node 1 has 3 possible configurations (disks). These partial graphs will be prioritized based on their estimated cost (Equation 3). This process will continue in a similar way. According to Rule 2, no extension will include Node 7 until a partial graph that contains all Nodes 3, 5 and 6 is dequeued. DAG* terminates only when an ANOC graph is dequeued.

Algorithm 1 provides the DAG* pseudocode. Initially, nodes (START, END) are inserted into the given graph $\mathcal{G}$ (Line 1), and the heuristics of each node are computed (Line 2) based on Equation 2. In Line 4 we insert a partial graph that contains the START node. Now, the loop in Line 7 contains the core idea of DAG*. In detail, the partial graph with the lowest estimated cost is dequeued. If the dequeued partial graph is an ANOC graph (Lines 9-12), then we will terminate our search returning the optimal solution (Line 22). This is because we will have a complete graph with lower real cost than any of the estimated (best case) costs of the (partial graphs) remaining in the queue (see Section III-E). If the dequeued partial graph was not an ANOC, then in Line 13 we find the nodes that are qualified for selection in the expansion of the dequeued partial graph based on Rule 2. In Line 14 we choose, among these candidates, one candidate node that will be inserted into the partial graph (Rule 1). Then, once we select the node that will be inserted, the inner loop (Line 16), for every available configuration (of the selected node), will (a) insert the node (assigned to a specific configuration) to the dequeued partial graph; (b) compute its estimated cost based on Equation 3; and (c) insert the extended partial graph to the priority queue in the appropriate index based on its estimated cost. Finally, the algorithm will proceed to the next iteration.

D. DAG* Search Complexity Analysis

Theorem 2. *Based on Rules 1 and 2, consider a topological order of all operators contained in any graph $\mathcal{G}$. In the worst-case scenario, the length of the priority queue right before the insertion of any node i, is equal to $num\_conf^{|\mathcal{X}(i)|}$, $\mathcal{X}(i)$ being the set of nodes that topologically precede $i \in \mathcal{G}$.*

Proof. We will prove this theorem by induction.

Base Step $i = 1$: Before inserting the first operator (non-auxiliary), based on our theorem the number of partial graphs that are contained in Q are $num\_conf^0 = 1$, which is true (remember that DAG* starts by inserting the START auxiliary node).

Inductive hypothesis: We assume that our theorem also holds when inserting operator k. As such, we take as granted that before inserting operator k the priority queue has a length of num_conf^{k-1}.

Inductive step: Now, before inserting operator $k+1$ to the partial graphs, we will have to expand all partial graphs that are contained in the priority queue at step k by inserting to each one of them all possible configurations of operator $k+1$. As such, the length of the queue after inserting operator $k+1$ will be equal to $num\_conf^{k-1} \cdot num\_conf = num\_conf^k$. Thus, we can see that our theorem also holds for step $k+1$.

So, we proved by induction that our theorem holds. □

Thus, in the worst case, the number of iterations-expansions of DAG* is $\sum_{i \in V_{\mathcal{G}}} num\_conf^{|\mathcal{X}(i)|} = \frac{num\_conf^{|V_{\mathcal{G}}|+1}-1}{num\_conf-1} = O(num\_conf^{|V_{\mathcal{G}}|})$. The running time cost of each expansion was presented in Section III.C. Also, in the worst case $O(num\_conf^{|V_{\mathcal{G}}|})$ partial graphs will be enqueued in the priority queue, but the memory footprint of each such partial graph is small, because we do not need to explicitly materialize them. Each partial graph can instead be represented by the IDs of the nodes that belong to it, along with a number for each node i.e., for the selected configuration for that node [21].

E. On DAG* Optimality

Note that due to the way we insert the auxiliary START, END nodes and since the solution to this optimization problem is an ANOC graph, it holds that $T_{\mathcal{G}^*}$ = END. However, as described in Section III-A, we know that $h(\text{END}) = 0$. As such, we can remove the second term of Equation 1, since it is always equal to zero due to the existing constraints for all valid solutions. Now, we prove theoretically that DAG* is able to discover the optimal graph for our optimization problem.

Theorem 3. *The ANOC graph, g, that DAG* returns is the one with the lowest cost, i.e., it is the optimal graph.*

Proof. Assume that some ANOC graph g' is the best one and not g, i.e., it holds that $Real(g') < Real(g)$. Initially, our approach begins by performing the extension procedure as described earlier. Now, consider the moment just before g

is chosen from the queue (dequeued). Some part of g' will also be on the queue; let's denote this partial graph as g''. Because g was "expanded" before g'' it holds that:

$$f(g) \leq f(g'') \Leftrightarrow Real(g) + h(g) \leq Real(g'') + h(g'') \quad (4)$$

Now, because g is an ANOC graph it holds that T_g = END and thus its heuristic is equal to zero, $h(g) = 0$. From Equation 4 we get:

$$Real(g) \leq Real(g'') + h(g'') \quad (5)$$

Moreover, we know that h is an admissible function. Thus, following our algorithm and Rules 1 and 2, we know that for any partial graph γ the following holds:

$$Real(\gamma) + h(\gamma) \leq Real(\forall \text{ANOC: Rule1} \wedge \text{Rule2} \wedge \text{contains}(\gamma)).$$

Therefore, it holds that:

$$Real(g'') + h(g'') \leq Real(g') \quad (6)$$

Finally, by combining Equations 5 and 6 we get the following:

$$Real(g) \leq Real(g'') + h(g'') \leq Real(g')$$

As a result, we have a *contradiction* and our hypothesis was wrong. Thus, ANOC graph g is the *optimal* graph. □

F. Dynamic Setting

Many real-world scenarios are characterized by dynamic changes that occur unpredictably over time. In IoT settings network devices may lose connectivity or new ones may subscribe to the network [9]. In the analytics scenarios considered in this work, the volume and velocity of continuously ingested data may vary over time [6]. Therefore, it is crucial for the employed algorithms to be able to adjust to these challenging settings by efficiently handling the changes that may alter configuration cost values. Given this motivation, we introduce an incremental version of DAG*, called d-DAG*.

To accommodate such dynamic cost changes efficiently, d-DAG* must undertake the task of updating its already-computed priority queue, Q, [3], ensuring the removal of nodes containing outdated information. Thus, given a change, we have to find the affected nodes and discard them from any partial graph $g \in Q$. In detail, the occurred change affects directly the node i, i.e., the node that its configuration(s) cost(s) just changed. As a result, the node i has to be discarded from any partial graph $g \in Q$. Moreover, since no partial graph will contain node i, due to Rule 2 (see Section III-C), d-DAG* has to also remove the nodes that exist in sets $Su(i), Su(Su(i)), \cdots$ to the END node, from all partial graphs in Q. On top of that, such change could also affect j's heuristic function, $h(j)$, where $j \in Pr(i)$ (see Equation 2). In such case, the change could similarly 'propagate' further, affecting the heuristics of the nodes in sets $Pr(Pr(i)), Pr(Pr(Pr(i))) \cdots$ to the START node. Once we check which of these nodes were affected, denoted as Aff, we also ensure that the post-change queue satisfies Lemma 1. As such, we discard all the nodes that exist in sets $Su(j), Su(Su(j)), \cdots$ to the END node, where $j \in Aff$.

Having discarded all the affected nodes from all partial graphs in Q, we compute the partial graphs estimated costs based on Equation 3. We then sort them with respect to the updated estimated costs and we remove the potential duplicates that may have been created. As a result of the aforementioned process, we have generated an updated priority queue by exploiting the information that had been already computed during the execution of DAG* before the change occurrence. Thus, we achieve the desired incremental version of our algorithm since our approach can exploit Q without necessarily starting from scratch to find the post-change optimal solution.

Algorithm 2 provides the d-DAG* pseudocode. Initially, we insert to the set of the affected nodes, denoted as Aff, the node i affected by the change (Line 4). In Line 5 we use a recursive function to find the nodes $Su(i), Su(Su(i)), \cdots$ and update Aff set. Then, if the occurred change affected the heuristic of node i we utilize a recursive function to find which of the nodes in $Pr(i), Pr(Pr(i)), \cdots$ were also affected and update Aff set (Lines 7 - 10). Now, given the Aff set, we have to discover the successors of j, where $j \in Aff$, in order to satisfy Lemma 1 and update the Aff set (Lines 11 - 13). Next, after the prescribed updates, the Aff set may contain duplicates which we can remove without affecting the result of the algorithm. Given the computed Aff set, we iterate through Q in order to remove from any stored partial graph the nodes that were affected and re-compute their estimated cost, which will be used as 'key' in order to sort the resulted priority queue Q_f (Lines 15 - 19). Once, we create the Q_f we remove any potential duplicate partial graphs that may have been created due to the node removal process described earlier. Finally, in Line 21, using the Q_f priority queue we call the Algorithm 1, i.e., the DAG* algorithm, which will not begin from scratch (an empty priority queue). In the worst case, d-DAG* will have to update all the affected partial graphs from the priority queue which are up to $num\_conf^{|V_G|}$ (see Section III-D); and call DAG* in Line 21 of Algorithm 2 with $Q_f = \emptyset$. Thus, the search will start from scratch and the complexity of d-DAG* will in the worst case be $O(num\_conf^{|V_G|})$.

G. On d-DAG*Optimality

Here we prove theoretically that d-DAG* is able to discover the optimal graph. For this section, we denote as Q the priority queue that DAG* has created in the initial setting, the first time it runs, and as Q_f the priority queue that d-DAG* generated after a change that occurred in the setting (Algorithm 2).

Lemma 4. *For any given partial graph g, if Lemma 1 is satisfied then we can always construct an ANOC graph, which contains the configurations of nodes that are already contained in g with any combination of configurations of the nodes that have not inserted to g yet.*

Proof. It is straightforward. □

[3]Note that initially the DAG* has been employed in an initial setting, returning the optimal graph through a priority queue Q that had been created (see Section III-C).

Algorithm 2 d-DAG* algorithm

Input: Graph $\mathcal{G}$, Priority queue Q, Configuration costs $conf_c$, change ch_i, old heuristic h, Priority queue $Q_f = \emptyset$
Output: Optimal graph $\mathcal{G}^*$

```
1: $h'$ = computeHeuristics($\mathcal{G}$, $conf_c$) {Update affected heuristics}
2: $Aff = \emptyset$
3: $i = ch_i$.getAffNode()
4: $Aff$.add($i$)
5: $su$ = findSuccessors($i$)
6: $Aff$.add($su$)
7: if $h(i)! = h'(i)$ then
8:     $pr$ = findPredecessors($i$)
9:     $Aff$.add($pr$)
10: end if
11: for $j \in Aff$ do
12:     $Aff$.add(findSuccessors($j$))
13: end for
14: $Aff$.removeDuplicates()
15: for $g \in Q$ do
16:     $g$.delete($Aff$)
17:     computeCost($g$)
18:     $Q_f$.enqueue($g$, sortKey = estimated cost)
19: end for
20: $Q_f$.removeDuplicates()
21: $\mathcal{G}^*$ = DAG*($Q_f$) (run Algorithm 1 from Line 3)
22: return $\mathcal{G}^*$
```

Lemma 5. *All partial graphs $g \in Q_f$ satisfy Lemma 1.*

Proof. It is straightforward. □

Based on the operation of DAG* (i.e., when it extends a node j it adds $|C_j|$ equal,[4] partial graphs to the queue, where each partial graph contains a different configuration of j), we can guarantee that at timestep t, where the optimal graph is dequeued, for each node i it holds that $\exists g \in Q : \forall c_k^i \in C_i$.

Now, assume that a change occurred in our setting that affected a set of nodes. Note that this set of nodes is computed by d-DAG* which creates the Q_f. In detail, d-DAG* does not delete any of the partial graphs stored in Q, but it replaces some of the partial graphs with one partial graph that does not contain any affected node. Thus, we cannot have a partial graph $g \in Q_f$ that contains only a subset of some node's configurations, because if a partial graph with a configuration $c_k^i \in C_i$ was affected, then all the 'equal' partial graphs of g that differ only to i's configurations will be affected as well. As a result $\forall i$ it holds either (a) $\exists g \in Q_f$ for $\forall c_k^i \in C_i$; or (b) $\nexists g \in Q_f$ for $\forall c_k^i \in C_i$.

Theorem 6. *Given Q_f there is no ANOC graph that cannot be reached.*

Proof. Assume that there is an ANOC graph g' which is unreachable. That means that in g' at least one pair of configurations $c_k^i \in C_i$ and $c_l^j \in C_j$ of nodes i and j cannot be constructed. We have the following three cases.

1) There is $\exists g_i, g_j \in Q_f$ that contains node i (with configuration c_k^i) and node j (with configuration c_l^j) accordingly.
   - If $g_i = g_j$. CONTRADICTION.
   - If $g_i \neq g_j$, then we know from Lemmas 4 and 5 that given any partial graph g in Q_f we can always reach an ANOC graph. So, by expanding g_i at some point we will reach node j and all of its available configurations. CONTRADICTION.
2) $\nexists g_i \in Q_f$ and $\nexists g_j \in Q_f$.
   - Given Lemmas 4 and 5 we know that we can reach an ANOC graph from any partial graph in Q_f. Thus by expanding any $g \in Q_f$ at some point, we will reach the node i and all of its available configurations. So consider the partial graph g_k that will be produced with configuration c_k^i when we expand g with node i. After some timesteps, by expanding the g_k partial graph at some point, we will reach the node j and all of its available configurations. CONTRADICTION.
3) $\exists g_i \in Q_f$ and $\nexists g_j \in Q_f$.
   - Given Lemmas 4 and 5 we know that we can reach an ANOC graph from any partial graph in Q_f. Thus by expanding g_i at some point, we will reach the node j and all of its available configurations. CONTRADICTION.

As a result we have a *contradiction* in every possible scenario and as such our hypothesis was wrong. Thus, d-DAG* can reach any possible ANOC graph. □

Given Theorems 3 and 6, by employing DAG*, not from scratch, but from Q_f as initial point, we will definitely reach the optimal solution.

Lemma 7. *The ANOC graph, g, that d-DAG* returns is the one with the lowest cost, i.e., it is the optimal graph.*

Proof. It is a direct consequence of Theorems 3 and 6. □

IV. DAG* over RapidMiner Studio

DAG* has been incorporated in a commercial platform, namely the streaming extension of RapidMiner Studio, running in the Streaming Optimization operator [30]. Cloud and network devices are registered as Connector objects in RapidMiner Studio, while the Streaming Optimization operator connects to the Optimization service via a separate connection object. Within the Streaming Optimization operator there is the Logical Workflow canvas where users can design their logical workflows, incorporating all the application logic but, being deprived from physical execution details. As soon as the user submits the designed workflow, the Streaming Optimization operator creates a JSON representation of the logical workflow and the network of devices and passes them to DAG* via a WebSocket. The Optimization service parses the JSON inputs and runs DAG*. It then responds back to the Studio with a new JSON describing the physical execution plan over the IoT network. The Studio renders the latter JSON into a graphical representation of the physical workflow, including separated workflow parts as Streaming Nest operators. Each Streaming Nest operator represents a separate job that is submitted for execution to the cloud or a network device, the corresponding part of the workflow has been assigned on. The

[4]We say that partial graphs g_1 and g_2 are equal if and only if they have the same heuristics, $V_{g_1} = V_{g_2}$, $E_{g_1} = E_{g_2}$ and all nodes in g_1 and g_2 have the same configuration except one node which has a different configuration in g_1 and g_2.

Workflow	Network of devices		
	num_conf = 7	num_conf = 15	num_conf = 31
TRAIN (8/4)	2401	50625	923521
PRED (9/4)	16807	759375	28629151
STATS (11/5)	117649	11390625	887503681
ETL (11/3)	5764801	2562890625	9.9E+35

TABLE I: Overview of setups in our experiments. Row headers are workflow names and (Total # operators/ # *cloud nodes*). Columns show number of devices/configurations in different networks; cells include # of possible solutions to the optimization problem.

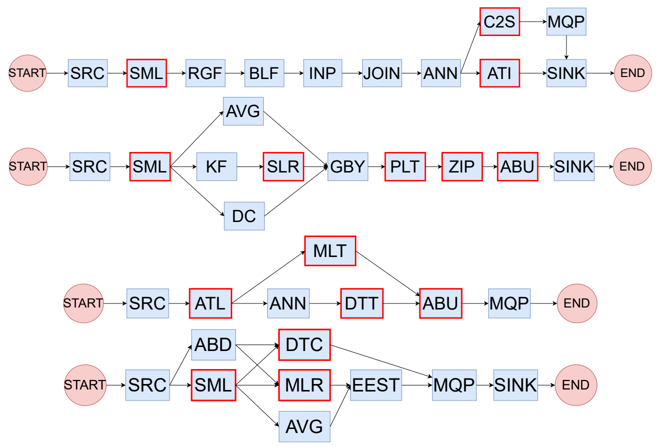


Fig. 2: The Extraction, Transform & Load (ETL – top), the Statistical Summarization (STATS – second), the Model Training (TRAIN – third) and the Predictive Analytics (PRED – bottom) workflows. The red borders in some operators represent the *cloud nodes* of each workflow. See [1] for the meaning of abbreviated operator names.

execution is monitored by the corresponding dashboard of each network participant. Remarkably, DAG* is a key enabler for the Streaming Extension of the Studio, since prior to DAG* the user had to either manually assign operators for execution in Streaming Nest operators or rely on the extremely high response times of an exhaustive search algorithm. Note though that in our experimental evaluation we choose standard IoT benchmarks and corresponding operators, instead of custom scenarios tailored to RapidMiner Studio, since DAG* can operate in any setting, detached from the Studio.

V. Experimental Evaluation

A. Experimental Setup

To evaluate DAG* we rely on the recent, highly cited and well documented RIoT benchmark [1] using four real-world workflows (termed TRAIN, STATS, ETL, PRED) from the smart cities domain [1], depicted in Figure 2. Each workflow consists of interconnected operators that are executed to accomplish a specific machine learning training, higher order statistic extraction, extract-transform-load and predictive analytics task, respectively. We monitor ground truth statistics (processing and communication latencies as costs for the benchmarked algorithms) over various networks of different sizes (small, medium, large), using the actively updated and highly cited iFogSim tool [31], [32] over these networks. We monitor setups consisting of num_conf = {7, 15, 31} configurations, i.e., devices that support the execution of an operator that is part of the workflow graph. Each pair of nodes and configurations has a different cost dependent on the computational complexity of the operator and the processing capacity of the network device. Some operators of each workflow cannot be executed on any device due to their nature (e.g. heavy duty machine learning operators [1]). Such operators are termed as *cloud nodes* since the cloud is their only valid configuration and they are highlighted via red border color for each workflow in Figure 2.

Given num_conf, we build corresponding binary tree network topologies. Processing latency statistics are composed of Raspberry Pi 4 devices running Apache Flink on Docker for the network side, as well as Apache Flink at the cloud side. The weights (network latencies) in the network links are in the interval $[1, 10]$, representing the cost of transmitting data from one node (or device) to its neighbors as assigned in iFogSim. The total cost (communication latency) of sending data from device d_1 to device d_2 is computed as the sum of the weights of the links that connects d_1 with d_2 in the network.

Table I summarizes the utilized setups. Rows correspond to workflows and their characteristics, while columns refer to the sizes (num_conf) of the used networks. Colored cells in the table show the number of possible solutions to the respective optimization problem (Equation 1). As Table I shows, we stress test DAG* in setups consisting from thousands to millions to billions and up to 9.9E+35 possible solutions. We present experiments on the challenging settings where $aggr = sum$ or $aggr = max$ in Equation 3.

Our comparisons are against the Spring Relaxation algorithm, termed `SpringRelax` in our plots, employed in state-of-the-art work of NEMO [11], against the `Exhaustive Search` algorithm and versus `Governor` [26]. DAG* and `Exhaustive Search` guarantee the optimality of the suggested workflow, while `SpringRelax` and `Governor` apply best effort heuristics. We choose two performance criteria. Aggregate (end-to-end) per tuple latency of the prescribed workflow execution plan devised by each algorithm, quantifying the produced plan's quality, and the number of iterations of each algorithm. `Exhaustive Search` and DAG* both produce the optimal plan, but DAG* examines a much smaller number of candidate plans, so we use the number of iterations to illustrate the effectiveness of DAG* to search a much smaller fraction of the search space. In `SpringRelax`, the number iterations is a parameter that should be set to define the repeated adjustments of each node's coordinates to better match the calculated distance between nodes (based on their positions in a virtual space) to the actual measured network distance in terms of latency. Each iteration calculates forces based on the error between the euclidean distance calculated from the virtual coordinates and the actual latency measured in the network. These forces pull

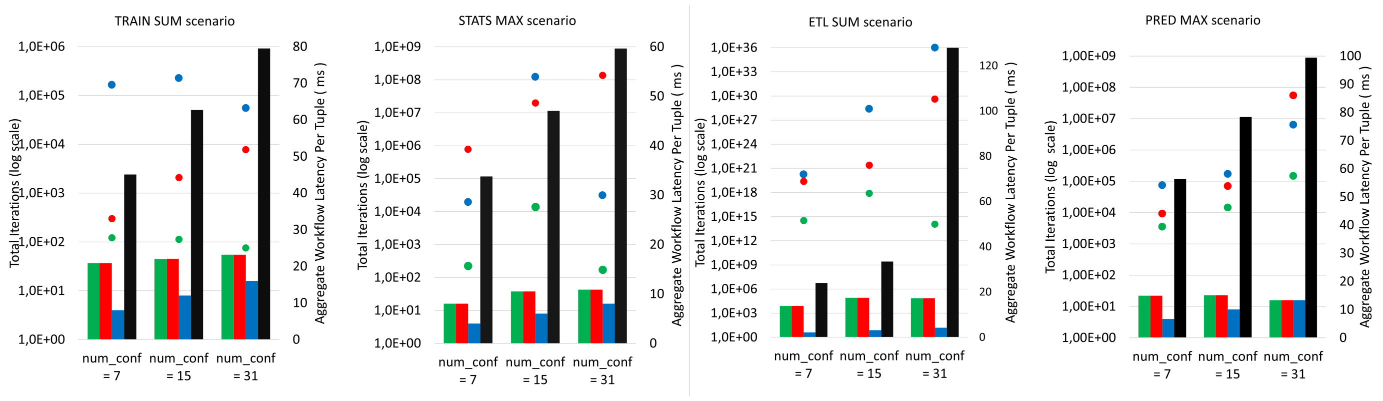


Fig. 3: DAG* efficiency across workflows and networks. Bars represent total number of iterations per algorithm (🟩 DAG*, 🟥 `SpringRelax`, 🟦 `Governor`, ⬛ `Exhaustive Search`). Dots plot the latency of the suggested workflow (● DAG* and `Exhaustive Search`, ● `SpringRelax`, ● `Governor`). The left vertical axis is in Log Scale showing the number of iterations per candidate algorithm. The right vertical axis corresponds to aggregate workflow latency per tuple for the devised plan.

or push operators to network devices to reduce the discrepancy. With more iterations, the nodes gradually settle into positions that represent the network's true layout, where calculated and actual distances closely align. Thus, iterations also account for the search space of `SpringRelax` only this time, the search space is a continuous one, with the possible positions of virtual space. Finally, `Governor`'s iterations account for finding the routing paths using DFS, based on the low latency policy [26].

B. *DAG* Results and Comparative Analysis*

In the first series of experiments, for each workflow, namely the TRAIN, STATS ETL and PRED workflows, we benchmark the efficiency of DAG* over three different-sized networks consisting of num_conf = {7, 15, 31} devices, accordingly. For instance, in the num_conf = 15 scenario, each of the nodes in a workflow can be executed on 15 different devices with different processing and communication latencies. Figure 3 presents the efficiency of DAG* compared to `SpringRelax`, `Governor` and `Exhaustive Search` across two (vertical in the plots) dimensions. The left vertical axis is the number of iterations of each algorithm, in Log Scale. The right vertical axis measures the aggregate (end-to-end) latency per tuple of the workflow suggested by the respective algorithm. The horizontal axis refers to the network sizes. We present scenarios where $aggr = sum$ for the ETL and TRAIN workflows, while $aggr = max$ for STATS, PRED. The trends are similar for the rest of the workflow, $aggr$ combinations.

Across all scenarios in Figure 3, DAG* returns the optimal graph with nearly 3 orders (TRAIN workflow - num_conf = 7) and up to 31 orders (ETL workflow - num_conf = 31) of magnitude fewer iterations compared to `Exhaustive Search`. Importantly, DAG* is not only efficient but also highly scalable since the number of iterations it requires shows slow incremental trends as the network size increases.

For a fair comparison with `SpringRelax`, we set its number of iterations equal to those of DAG* in each setup, therefore the corresponding bars in the plots are of equal height. The overall latency of the suggested workflow by DAG* and `SpringRelax` are represented as dots, the values of which correspond to the rightmost vertical axis (green dot for DAG* and red dot for `SpringRelax`). We can observe that DAG* consistently outperforms `SpringRelax` in terms of the overall latency of the suggested workflow, since `SpringRelax` outputs workflows that are from 20% (TRAIN workflow - num_conf = 7) and up to 3.6 times (STATS workflow - num_conf = 31) worse compared to DAG*. We further note that these scenarios are in favor of `SpringRelax`. For instance, if we increase the number of `SpringRelax` iterations in the ETL scenario (Figure 3), the latency of `SpringRelax`-suggested workflow never improves. On the other hand, if we decrease the number of iterations of `SpringRelax` in the same scenario, the suggested workflow can exhibit up to -60% worst overall latency performance. In other words, `SpringRelax` seems to achieve its best performance around a number of iterations that is equivalent with DAG*, but its best effort heuristic does not accurately approximate the optimal solution.

Governor shows a reduced number of iterations because, besides the cloud nodes (heavy duty operators executable only at the cloud) of the workflows, it also manually fixes non-blocking operators (e.g. filters) close to the data sources and sinks on the cloud. `Governor` shows the worst performance in terms of the latency of its devised plans across competitors, with only few cases of the STATS, num_conf = {7, 31} and PRED, num_conf = 31 workflows where it outperforms `SpringRelax`. Still `Governor` plans' latency is considerably higher compared to the optimal DAG* in all cited scenarios.

C. *Varying Network Organizations*

We now evaluate DAG* and its competitors under two different, alternative network settings: (i) a Star topology where the Raspberry Pis are directly connected to the cloud and (ii) the NES-like topology of [7], where the fog layer connects Raspberry Pis to routing nodes (switches and routers). Routing nodes have no data processing capacity (thus, not counted in num_conf), but offer high-speed data transfer and, therefore, roughly 10x reduced network latency. We indicatively focus on

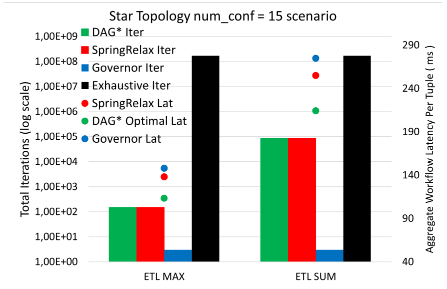


Fig. 4: Star topology results.

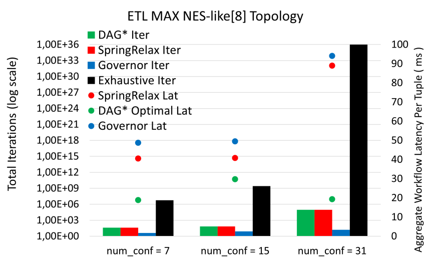
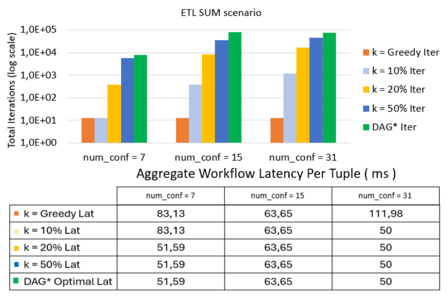


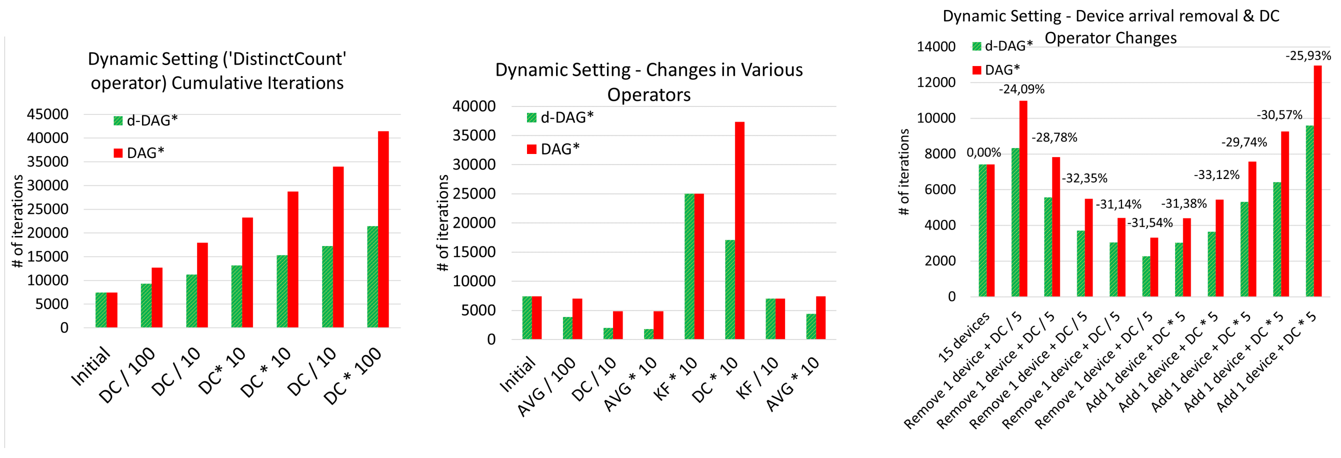
Fig. 5: NES-like topology [7] results.



Aggregate Workflow Latency Per Tuple (ms)

	num_conf = 7	num_conf = 15	num_conf = 31
k = Greedy Lat	83,13	63,65	111,98
k = 10% Lat	83,13	63,65	50
k = 20% Lat	51,59	63,65	50
k = 50% Lat	51,59	63,65	50
DAG* Optimal Lat	51,59	63,65	50

Fig. 6: Results of Approximate DAG*.



(a) (b) (c)

Fig. 7: Results on the dynamic setting for the STATS workflow (see also Figure 1), with $aggr = sum$ and $num\_conf = 15$. The x-axis indicates cost fluctuations and the name of the operator in which they occur. The y-axis shows cumulative iterations in Figure 7a and number of iterations per change (Figures 7b, 7c) for DAG* and d-DAG*. DC stands for Distinct Count, AVG for Average and KF for the Kalman filter operators. Both DAG* and d-DAG* return an execution plan of optimal latency.

the ETL workflow, but the results are analogous for the rest of the workflows cited in Figure 2. In Figure 4, for $num\_conf =$ 15, we observe that DAG* provides 16% to 20% better latency in the workflow it prescribes for execution, compared to the one prescribed by `SpringRelax`. Moreover, `Governor` has limited routing path options in the Star topology, therefore, it requires fewer iterations, as shown in Figure 4. However, `Governor` also exhibits the worse latency performance among all competitors, falling up to ~25% short (both for $aggr = \{max, sum\}$), compared to DAG*. Similarly, in the NES-like topology of Figure 5 for $aggr = max$, we observe that DAG* provides from 2x ($num\_conf = \{7,15\}$) and up to 4.5x ($num\_conf = 31$) improved latency compared to the second best approach of `SpringRelax`, surpassing the 3.5x improvement observed in Section V-B.

Among the ETL workflow experiments in Figure 4 and those in Figure 3, we observe that the latency of the plans devised by each and every competitor are up to four times higher in the Star topology. This validates the claim made in the Section I that multilayer IoT organizations delegating tasks across edge, fog, cloud layers reduces raw data volumes sent to the cloud and lowers transmission costs, while also exploiting the processing power across the cloud to edge continuum.

D. *d-DAG* and Dynamic Settings*

We proceed by studying the increased efficiency of d-DAG* in the dynamic setting. We stress that DAG* (running from scratch) and the incremental d-DAG* will always output the optimized workflow with the optimal latency (Theorem 3 and Lemma 7), with d-DAG* achieving this outcome in a reduced number of iterations due to exploiting the priority queue from prior optimizations. We indicatively provide results on the STATS workflow with $num\_conf = 15$, but the trends are the same for the other workflows. We examine various cases of dynamic scenarios. Initially, we have two cases of dynamic scenarios where network costs change due to volatile stream properties. In the first scenario, we monitor the cost of the same (`DistinctCount`) operator which continuously fluctuates due to changes in the amount of data we ingest from `Source` in Figure 1. The horizontal axis of Figure 7a shows the series of fluctuations `DistinctCount` experiences. When the cost of an operator increases or decreases by a factor of 10 or more, we rerun our DAG* (red bar) and d-DAG* (green bar). For instance, the horizontal axis of Figure 7a shows that after the initial setup, `DistinctCount`'s cost reduces by a factor of 100 and then it further decreases by a factor of 10. After two steps, it recovers its initial cost, before having a 10-fold cost drop. Finally, its cost increases by a factor of 100. The vertical axis in Figure 7a shows the total, cumulative number of iterations DAG* and d-DAG* required to compute the new optimal graph up to the point the respective change happened, given the previous ones. Overall, d-DAG* improves DAG* by ~48% (last pair of bars in Figure 7a).

In the second scenario we monitor the changes that happen in the entire workflow and we rerun d-DAG* and DAG* in case any operator shows 10-fold cost fluctuations. In Figure 7b the horizontal axis again shows cost fluctuations and the operator they occur, but this time the vertical axis does not show the cumulative, but the individual number of iterations d-DAG* and DAG* required to compute the new optimal graph each time. We choose to do so to better illustrate the effect of a cost change in conjunction with the position of the operator in the workflow of Figure 1. We highlight that the equality in the number of iterations when `KalmanFilter` costs change is because, due to a change, all operators are affected (see Section III-F) and thus d-DAG* discards all partial graphs from the priority queue and practically starts from scratch. In 5 out of the 7 other cases in Figure 7b, d-DAG* considerably improves DAG* with median ~43% and maximum 64% reduction in iterations.

To stress test d-DAG*, in Figure 7c, we change the cost of an operator on par with devices departing from or registering to the network. We begin with num_conf=15 and progressively one device departs causing the cost of the Distinct Count (DC in the figure) a 5-fold cost reduction. This happens progressively five times. Thus, when the experiment reaches the middle bar of Figure 7c, num_conf has been reduced by 5 nodes, i.e., lost 1/3 of its devices. The reverse change happens with the progressive addition of 5 devices, one by one, and corresponding 5-fold cost increments for the Distinct Count operator. Overall, d-DAG* preserves its ability to improve DAG*, saving between 24% and 33% iterations.

E. Adversarial Scenarios

Adversarial scenarios involve setups where DAG* needs more iterations to find the optimal graph. The first adversarial scenario is when all nodes possess configurations with uniform cost values. In such cases, the partial graphs in the priority queue will have similar estimated costs during the iterations of the algorithm, and many partial graphs will be expanded one after the other instead of having few promising partial graphs being prioritized for expansion. This phenomenon, in turn, delays the extraction of the optimal graph. To attest for that scenario we create a setup with the STATS workflow with num_conf=15 and we assign to all configurations of all nodes uniform cost values $U(0, 10)$, fixing all edge weights to the middle of that interval, i.e., communication latencies are fixed to 5. We repeat the experiment 5 times (Rep#1 to Rep#5) and we measure DAG* iterations. As Table II shows, DAG* saves 2 orders of magnitude per experiment and on average compared to `Exhaustive Search`, instead of 3 or more orders of magnitude in Figure 3. The `SpringRelax` does not change for equivalent number of iterations to those in Figure 3 and becomes worse when we set its number of iterations equal to DAG*, due to the fact that it tends to accumulate more operators at the cloud side.

	DAG*	**Exhaustive**
Rep#1	104893	11 Million
Rep#2	105166	11 Million
Rep#3	43818	11 Million
Rep#4	104960	11 Million
Rep#5	104898	11 Million
Average	92747	11 Million

TABLE II: #Iterations, Adversarial Scenario of~U(0,10) costs.

The second adversarial scenario involves d-DAG*. Recall that *cloud nodes* have a single configuration. If a new cloud configuration is added for them and simultaneously entails reduced cost for every cloud node, then the heuristic cost and the real cost of the workflow need to be recomputed. Therefore, d-DAG* will require the same number of iterations as DAG*, because all partial graphs in the queue will be affected. We emphasize, though that still DAG* will explore only a small portion of the search space. Due to space constraints we omit the corresponding graph, but we note that d-DAG* and DAG* will coincide in every such scenario, however, still saving orders of magnitude iterations.

F. DAG* Approximate Versions

One can loosen the optimality requirement and run approximate versions of DAG* to further reduce the explored search space. The basic concept of Approximate DAG* is to restrict the allowed size of the priority queue. Consider that we have num_conf possible configurations for every node of the workflow (Figure 1). Further consider a parameter $\varkappa$ destined to restrict the number of partial graphs that will be enqueued upon a partial graph expansion (Section III-C). If we set $\varkappa = \xi\%$, this means that every time we dequeue a partial graph and we expand it based on Rule 1 and Rule 2, we keep and choose to enqueue only the resulted $\lceil \xi\% \cdot num\_conf \rceil$ new partial graphs with the lowest estimated cost. For instance, if we set $\varkappa = 1\%$ this would mean that upon dequeuing a partial graph and extend it, only the best among the new, expanded partial graphs will be enqueued and the rest will be discarded. Therefore, setting $\varkappa = 1\%$ reduces DAG* to a Greedy approach which chooses only the best configuration per operator. Similarly, for a network of $num\_conf = 15$ and $\varkappa = 10\%$, we have $\lceil \varkappa\% \cdot num\_conf \rceil = 2$ meaning that upon dequeuing a partial graph and extend it, only the two best among the new, expanded partial graphs will be enqueued.

In Figure 6, we plot the number of iterations of DAG* along with the latency table for the suggested workflow execution graph, for $\varkappa \in \{1\%(\text{Greedy}), 10\%, 20\%, 50\%\}$. For our comparative analysis, we further include DAG* in the bar chart. In Figure 6, we can see that for smaller network sizes and search spaces (num_conf=7 – also see Table I), irrespectively of $\varkappa$, the quality (latency value in the table under the plot) of the solution of the Approximate DAG* is 38% worse compared to the exact DAG*. This is not too great a deficiency because the exact version of DAG* already has pruned a large portion of the search space requiring only few tens of iterations for outputting the optimal graph.

On the contrary, for medium network sizes and search spaces (*num_conf*=15 – also see Table I), Figure 6 shows that all the Approximate DAG* versions provide the optimal solutions (latency value of 63.65 in the table under the plot). Based on this, one can utilize the Greedy version of the Approximate DAG* to reduce the amount of explored search space to only a dozen of options. For large network sizes and search spaces (*num_conf*=31 – also see Table I) only the Greedy version of DAG* fails to provide a decent solution, suggesting a workflow execution with more than 2 times worse latency (111.98ms vs the optimal 50ms in the latency table under the plot), while all the other approximate versions of DAG* coincide with the optimal latency of the workflow devised by DAG*. This means that an approximate version with $\varkappa$ = 10% can further reduce the algorithm iterations by almost two orders of magnitude compared to the optimal DAG*, without practically compromising the optimality of the output workflow. In Figure 6, we test Approximate DAG* on the ETL workflow, but we stress that these trends are exactly the same for the other (STATS, TRAIN, PRED) workflows.

Approximate DAG* expectedly makes sense for medium to very large search spaces. This conclusion is drawn from Figure 6 where *num_conf*= 7 is more affected by the approximation for $\varkappa$ = Greedy and $\varkappa$ = 10%, because the latency of the devised plan (first column, first and second rows of the table in Figure 6) is higher compared to DAG* Optimal Lat (first column, last row of the table in Figure 6) for the same *num_conf* = 7. In the higher *num_conf*s cited in the table, any approximation with $\varkappa \geq 10\%$ achieves the optimal latency, equal to DAG* Optimal Lat of each column.

VI. Related Work

Workflow Optimization: Early works on optimal operator or workflow execution arose with the development of the first stream processing systems [12], [13] aiming at optimizing query execution over a single server, with no focus on network performance. Later on, approaches such as Medusa [14] and SQPR [16] addressed network related metrics on par with efficient execution per host, the focus being to primarily balance the load and minimize resource usage among networked hosts.

A series of works optimize the execution of streaming analytics workflows at the cloud over powerful Big Data frameworks [22], [23], [33]. Such approaches are impractical for IoT computing applications due to their excessive complexity on large networks and their inability to address changes through efficient, incremental re-optimizations [11]. DAG* addresses these issues via the powerful admissible heuristic and via the incremental nature of d-DAG*. Cross-platform optimization [5], [6], [17]–[20] accounts for optimizing workflows at the cloud side and not necessarily in the scope of a streaming setting. Relevant frameworks [17]–[20] focus on a single cloud, running multiple Big Data platforms, thus not accounting for multiple hosts or efficient re-optimization.

The prominent work of SBON [15] was among the first to account for network resource utilization for network aware operator placement. Rizou et al [10] complemented the SBON approach to multiple operator placements. Both approaches account only for network performance measures, i.e., network latency, without examining the processing latency of devices.

The spring relaxation algorithm used in NEMO [11] is also utilized in SBON and corresponds to the `SpringRelax` candidate in our plots. DAG* outperforms spring relaxation and does not require specific assumptions on the organization of the IoT network, while NEMO operates over clustered network architectures. Tzortzi et al [9] employ machine learning for operator and network cost estimation. Learning such metrics requires a daunting training phase which is infeasible for large scale and volatile environments. In contrast, our experimentation relied on obtaining cost estimations via a well-established IoT simulator [31], [32]. The Governor approach [26], used in our experimental evaluation, plans workflow execution across IoT via user-defined policies.

Finally, the works of Akili et al [34], [35] explore in network execution of query patterns for complex event recognition.

A* Algorithm and Variants: The A* [27], [36] inspiring DAG* conceptualization is a well-known algorithm that stands out for its efficiency in finding the shortest path between two nodes in a graph. A plethora of A* variants have been developed [28]. The Iterative Deepening A* (IDA*) algorithm [36] is a memory-efficient variant of A*. The D* algorithm [37] is a variation of A* designed to efficiently find the shortest path in a dynamic environment where the cost or availability of edges change over time. The Lifelong Planning A* (LPA*) algorithm [38] is another incremental version of A* designed to address changing edge costs or dynamic graph structures. The, also incremental, D* Lite [38] combines the virtues of D* and LPA*. All these algorithms return optimal path instead of optimal graph, therefore not being suitable for our setting.

VII. Conclusions & Future Work

We present a novel A*-alike algorithm for optimal workflow graph discovery. Our algorithm is of high value for task allocation and operator placement in a variety of streaming analytics optimization scenarios arising in a wide spectrum of IoT domains, indicatively including smart city, smart factory, smart grid and robotics applications. We detail the new DAG* algorithm and an incremental d-DAG* version of it for dynamic environments. The novel algorithm prunes large parts of the search space, from nearly 3 and up to 31 orders of magnitude in our experiments. Importantly, it does so, simultaneously guaranteeing optimal graph solutions. It further outperforms heuristics used in recent, state of the art techniques, in terms of the efficiency of the suggested workflow. d-DAG* further boosts the efficiency of DAG* halving the number of algorithm iterations. Our current focus is on constant factor approximations of DAG*'s optimal solutions and on parallel versions of it for enhanced performance.

References

[1] A. Shukla, S. Chaturvedi, and Y. Simmhan, "Riotbench: An iot benchmark for distributed stream processing systems," *Concurrency and Computation: Practice and Experience*, vol. 29, no. 21, p. e4257, 2017, e4257 cpe.4257. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.4257

[2] A. P. Lepping, H. M. Pham, L. Mons, B. Rueb, P. M. Grulich, A. Chaudhary, S. Zeuch, and V. Markl, "Showcasing data management challenges for future iot applications with nebulastream," *Proc. VLDB Endow.*, vol. 16, no. 12, pp. 3930–3933, 2023. [Online]. Available: https://www.vldb.org/pvldb/vol16/p3930-lepping.pdf

[3] F. Siqueira and J. G. Davis, "Service computing for industry 4.0: State of the art, challenges, and research opportunities," *ACM Comput. Surv.*, vol. 54, no. 9, oct 2021. [Online]. Available: https://doi.org/10.1145/3478680

[4] S. Zeuch, A. Chaudhary, B. D. Monte, H. Gavriilidis, D. Giouroukis, P. M. Grulich, S. Breß, J. Traub, and V. Markl, "The nebulastream platform for data and application management in the internet of things," in *10th Conference on Innovative Data Systems Research, CIDR 2020, Amsterdam, The Netherlands, January 12-15, 2020, Online Proceedings*. www.cidrdb.org, 2020. [Online]. Available: http://cidrdb.org/cidr2020/papers/p7-zeuch-cidr20.pdf

[5] N. Giatrakos, E. Alevizos, A. Deligiannakis, R. Klinkenberg, and A. Artikis, "Proactive streaming analytics at scale: A journey from the state-of-the-art to a production platform," in *Proceedings of the 32nd ACM International Conference on Information and Knowledge Management, CIKM 2023, Birmingham, United Kingdom, October 21-25, 2023*. ACM, 2023, pp. 5204–5207. [Online]. Available: https://doi.org/10.1145/3583780.3615293

[6] N. Giatrakos and et al, "Infore: Interactive cross-platform analytics for everyone," in *CIKM '20: The 29th ACM International Conference on Information and Knowledge Management, Virtual Event, Ireland, October 19-23, 2020*. ACM, 2020, pp. 3389–3392.

[7] S. Zeuch, A. Chaudhary, B. D. Monte, H. Gavriilidis, D. Giouroukis, P. M. Grulich, S. Breß, J. Traub, and V. Markl, "The nebulastream platform for data and application management in the internet of things," 2020. [Online]. Available: http://cidrdb.org/cidr2020/papers/p7-zeuch-cidr20.pdf

[8] J. Dongarra, B. Tourancheau, D. Balouek-Thomert, E. G. Renart, A. R. Zamani, A. Simonet, and M. Parashar, "Towards a computing continuum: Enabling edge-to-cloud integration for data-driven workflows," *Int. J. High Perform. Comput. Appl.*, vol. 33, no. 6, p. 1159–1174, nov 2019. [Online]. Available: https://doi.org/10.1177/1094342019877383

[9] M. Tzortzi, C. Kleitsikas, A. Politis, S. Niarchos, K. Doka, and N. Koziris, "Planning workflow executions over the edge-to-cloud continuum," in *Algorithmic Aspects of Cloud Computing - 8th International Symposium, ALGOCLOUD 2023, Amsterdam, The Netherlands, September 5, 2023, Revised Selected Papers*, ser. Lecture Notes in Computer Science, vol. 14053. Springer, 2023, pp. 9–24. [Online]. Available: https://doi.org/10.1007/978-3-031-49361-4_1

[10] S. Rizou, "Concepts and algorithms for efficient distributed processing of data streams," Ph.D. dissertation, University of Stuttgart, 2014. [Online]. Available: http://elib.uni-stuttgart.de/opus/volltexte/2014/8835/

[11] X. Chatziliadis, E. T. Zacharatou, A. Eracar, S. Zeuch, and V. Markl, "Efficient placement of decomposable aggregation functions for stream processing over large geo-distributed topologies," *Proc. VLDB Endow.*, vol. 17, no. 6, pp. 1501–1514, 2024. [Online]. Available: https://www.vldb.org/pvldb/vol17/p1501-chatziliadis.pdf

[12] M. A. Shah, J. M. Hellerstein, S. Chandrasekaran, and M. J. Franklin, "Flux: An adaptive partitioning operator for continuous query systems," in *Proceedings of the 19th International Conference on Data Engineering, March 5-8, 2003, Bangalore, India*, U. Dayal, K. Ramamritham, and T. M. Vijayaraman, Eds. IEEE Computer Society, 2003, pp. 25–36. [Online]. Available: https://doi.org/10.1109/ICDE.2003.1260779

[13] D. J. Abadi, Y. Ahmad, M. Balazinska, U. Çetintemel, M. Cherniack, J. Hwang, W. Lindner, A. Maskey, A. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, and S. B. Zdonik, "The design of the borealis stream processing engine," in *Second Biennial Conference on Innovative Data Systems Research, CIDR 2005, Asilomar, CA, USA, January 4-7, 2005, Online Proceedings*. www.cidrdb.org, 2005, pp. 277–289. [Online]. Available: http://cidrdb.org/cidr2005/papers/P23.pdf

[14] M. Balazinska, H. Balakrishnan, and M. Stonebraker, "Contract-based load management in federated distributed systems," in *1st Symposium on Networked Systems Design and Implementation (NSDI 2004), March 29-31, 2004, San Francisco, California, USA, Proceedings*, R. T. Morris and S. Savage, Eds. USENIX, 2004, pp. 197–210. [Online]. Available: http://www.usenix.org/events/nsdi04/tech/balazinska.html

[15] P. R. Pietzuch, J. Ledlie, J. Shneidman, M. Roussopoulos, M. Welsh, and M. I. Seltzer, "Network-aware operator placement for stream-processing systems," in *Proceedings of the 22nd International Conference on Data Engineering, ICDE 2006, 3-8 April 2006, Atlanta, GA, USA*, L. Liu, A. Reuter, K. Whang, and J. Zhang, Eds. IEEE Computer Society, 2006, p. 49. [Online]. Available: https://doi.org/10.1109/ICDE.2006.105

[16] E. Kalyvianaki, W. Wiesemann, Q. H. Vu, D. Kuhn, and P. Pietzuch, "Sqpr: Stream query planning with reuse," in *Proceedings of the 2011 IEEE 27th International Conference on Data Engineering*, ser. ICDE '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 840–851. [Online]. Available: http://dx.doi.org/10.1109/ICDE.2011.5767851

[17] K. Doka, N. Papailiou, D. Tsoumakos, C. Mantas, and N. Koziris, "Ires: Intelligent, multi-engine resource scheduler for big data analytics workflows," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, T. K. Sellis, S. B. Davidson, and Z. G. Ives, Eds. ACM, 2015, pp. 1451–1456. [Online]. Available: https://doi.org/10.1145/2723372.2735377

[18] I. Gog, M. Schwarzkopf, N. Crooks, M. P. Grosvenor, A. Clement, and S. Hand, "Musketeer: all for one, one for all in data processing systems," in *Proceedings of the Tenth European Conference on Computer Systems, EuroSys 2015, Bordeaux, France, April 21-24, 2015*, L. Réveillère, T. Harris, and M. Herlihy, Eds. ACM, 2015, pp. 2:1–2:16. [Online]. Available: https://doi.org/10.1145/2741948.2741968

[19] J. Duggan, A. J. Elmore, M. Stonebraker, M. Balazinska, B. Howe, J. Kepner, S. Madden, D. Maier, T. Mattson, and S. B. Zdonik, "The bigdawg polystore system," *SIGMOD Rec.*, vol. 44, no. 2, pp. 11–16, 2015. [Online]. Available: https://doi.org/10.1145/2814710.2814713

[20] D. Agrawal, S. Chawla, B. Contreras-Rojas, A. K. Elmagarmid, Y. Idris, Z. Kaoudi, S. Kruse, J. Lucas, E. Mansour, M. Ouzzani, P. Papotti, J. Quiané-Ruiz, N. Tang, S. Thirumuruganathan, and A. Troudi, "RHEEM: enabling cross-platform data processing - may the big data be with you! -," *Proc. VLDB Endow.*, vol. 11, no. 11, pp. 1414–1427, 2018. [Online]. Available: http://www.vldb.org/pvldb/vol11/p1414-agrawal.pdf

[21] Z. Kaoudi, J. Quiané-Ruiz, B. Contreras-Rojas, R. Pardo-Meza, A. Troudi, and S. Chawla, "Ml-based cross-platform query optimization," in *36th IEEE International Conference on Data Engineering, ICDE 2020, Dallas, TX, USA, April 20-24, 2020*. IEEE, 2020, pp. 1489–1500. [Online]. Available: https://doi.org/10.1109/ICDE48307.2020.00132

[22] S. Venkataraman, A. Panda, K. Ousterhout, M. Armbrust, A. Ghodsi, M. J. Franklin, B. Recht, and I. Stoica, "Drizzle: Fast and adaptable stream processing at scale," in *Proceedings of the 26th Symposium on Operating Systems Principles, Shanghai, China, October 28-31, 2017*. ACM, 2017, pp. 374–389. [Online]. Available: https://doi.org/10.1145/3132747.3132750

[23] V. Cardellini, F. L. Presti, M. Nardelli, and G. R. Russo, "Optimal operator deployment and replication for elastic distributed data stream processing," *Concurr. Comput. Pract. Exp.*, vol. 30, no. 9, 2018. [Online]. Available: https://doi.org/10.1002/cpe.4334

[24] U. Srivastava, K. Munagala, and J. Widom, "Operator placement for in-network stream query processing," in *Proceedings of the Twenty-fourth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 13-15, 2005, Baltimore, Maryland, USA*, C. Li, Ed. ACM, 2005, pp. 250–258. [Online]. Available: https://doi.org/10.1145/1065167.1065199

[25] G. Chatzimilioudis, A. Cuzzocrea, D. Gunopulos, and N. Mamoulis, "A novel distributed framework for optimizing query routing trees in wireless sensor networks via optimal operator placement," *J. Comput. Syst. Sci.*, vol. 79, no. 3, pp. 349–368, 2013. [Online]. Available: https://doi.org/10.1016/j.jcss.2012.09.013

[26] A. Chaudhary, S. Zeuch, and V. Markl, "Governor: Operator placement for a unified fog-cloud environment," in *Proceedings of the 23rd International Conference on Extending Database Technology, EDBT 2020, Copenhagen, Denmark, March 30 - April 02, 2020*, A. Bonifati, Y. Zhou, M. A. V. Salles, A. Böhm, D. Olteanu, G. H. L. Fletcher,

A. Khan, and B. Yang, Eds. OpenProceedings.org, 2020, pp. 631–634. [Online]. Available: https://doi.org/10.5441/002/edbt.2020.81

[27] P. Hart, N. Nilsson, and B. Raphael, "A formal basis for the heuristic determination of minimum cost paths," *IEEE Transactions on Systems Science and Cybernetics*, vol. 4, no. 2, pp. 100–107, 1968. [Online]. Available: https://doi.org/10.1109/tssc.1968.300136

[28] D. Foead, A. Ghifari, M. B. Kusuma, N. Hanafiah, and E. Gunawan, "A systematic literature review of a* pathfinding," *Procedia Computer Science*, vol. 179, pp. 507–514, 2021, 5th International Conference on Computer Science and Computational Intelligence 2020. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1877050921000399

[29] "RapidMiner Streaming Extension," https://marketplace.rapidminer.com/UpdateServer/faces/product_details.xhtml?productId=rmx_streaming. The latest code will be provided open-source should this paper get accepted.

[30] YouTube, "DAG* Demo," 2024, accessed: 2024-10-25. [Online]. Available: https://www.youtube.com/watch?v=CsQWz-8F7a4

[31] H. Gupta, A. Vahid Dastjerdi, S. K. Ghosh, and R. Buyya, "ifogsim: A toolkit for modeling and simulation of resource management techniques in the internet of things, edge and fog computing environments," *Software: Practice and Experience*, vol. 47, no. 9, pp. 1275–1296, 2017. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.2509

[32] M. R. Mahmud, S. Pallewatta, M. Goudarzi, and R. Buyya, "ifogsim2: An extended ifogsim simulator for mobility, clustering, and microservice management in edge and fog computing environments," *J. Syst. Softw.*, vol. 190, p. 111351, 2022. [Online]. Available: https://doi.org/10.1016/j.jss.2022.111351

[33] A. Agrawal and A. Floratou, "Dhalion in action: Automatic management of streaming applications," *Proc. VLDB Endow.*, vol. 11, no. 12, pp. 2050–2053, 2018. [Online]. Available: http://www.vldb.org/pvldb/vol11/p2050-agrawal.pdf

[34] S. Akili and M. Weidlich, "Muse graphs for flexible distribution of event stream processing in networks," in *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*, G. Li, Z. Li, S. Idreos, and D. Srivastava, Eds. ACM, 2021, pp. 10–22. [Online]. Available: https://doi.org/10.1145/3448016.3457318

[35] S. Akili, S. Purtzel, and M. Weidlich, "Inev: In-network evaluation for event stream processing," *Proc. ACM Manag. Data*, vol. 1, no. 1, pp. 101:1–101:26, 2023. [Online]. Available: https://doi.org/10.1145/3588955

[36] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, 3rd ed. Prentice Hall, 2010.

[37] A. Stentz, "The focussed d* algorithm for real-time replanning," in *Proceedings of the 14th International Joint Conference on Artificial Intelligence - Volume 2*, ser. IJCAI'95. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1995, p. 1652–1659.

[38] D. F. Sven Koenig, Maxim Likhachev, "Lifelong planning $A^\star$," *Artificial Intelligence*, vol. 155, no. 1, pp. 93–146, 2004.