http://doi.org/10.35784/iapgos.5858 *received: 02.02.2024 | revised: 20.02.2024 | accepted: 18.03.2024 | available online: 31.03.2024*

RUNNING A WORKFLOW WITHOUT WORKFLOWS: A BASIC ALGORITHM FOR DYNAMICALLY CONSTRUCTING AND TRAVERSING AN IMPLIED DIRECTED ACYCLIC GRAPH IN A NON-DETERMINISTIC ENVIRONMENT

Fedir Smilianets, Oleksii Finogenov
National Technical University of Ukraine "Igor Sikorsky Kyiv Polytechnic Institute", Kyiv, Ukraine

Abstract. *This paper introduces a novel algorithm for dynamically constructing and traversing Directed Acyclic Graphs (DAGs) in workflow systems, particularly targeting distributed computation and data processing domains. Traditional workflow management systems rely on explicitly defined, rigid DAGs, which can be cumbersome to maintain, especially in response to frequent changes or updates in the system. Our proposed algorithm circumvents the need for explicit DAG construction, instead opting for a dynamic approach that iteratively builds and executes the workflow based on available data and operations, through a combination of entities like Data Kinds, Operators, and Data Units, the algorithm implicitly forms a DAG, thereby simplifying the process of workflow management. We demonstrate the algorithm's functionality and assess its performance through a series of tests in a simulated environment. The paper discusses the implications of this approach, especially focusing on cycle avoidance and computational complexity, and suggests future enhancements and potential applications.*

Keywords: distributed computing, directed acyclic graph, pipeline processing

OBLICZANIE PRZEPŁYWÓW PRACY BEZ PRZEPŁYWÓW PRACY: PODSTAWOWY ALGORYTM DYNAMICZNEGO KONSTRUOWANIA I PRZESZUKIWANIA NIEJAWNEGO SKIEROWANEGO GRAFU ACYKLICZNEGO W ŚRODOWISKU NIEDETERMINISTYCZNYM

Streszczenie. *W artykule przedstawiono nowy algorytm dynamicznego konstruowania i przejść skierowanych grafów acyklicznych (DAG) w systemach zarządzania przepływem pracy, w szczególności tych ukierunkowanych na domeny obliczeń rozproszonych i przetwarzania danych. Tradycyjne systemy zarządzania przepływem pracy opierają się na jawnie zdefiniowanych, sztywnych grafach DAG, które mogą być uciążliwe w utrzymaniu, zwłaszcza w odpowiedzi na częste zmiany lub aktualizacje systemu. Proponowany algorytm pozwala uniknąć konieczności jawnego konstruowania SAG, zamiast tego wybierając dynamiczne podejście, które iteracyjnie buduje i wykonuje przepływy pracy w oparciu o dostępne dane i operacje. Korzystając z kombinacji jednostek, takich jak typ danych, operator i element danych, algorytm niejawnie buduje DAG, upraszczając w ten sposób proces zarządzania przepływami pracy. Demonstrujemy funkcjonalność algorytmu i oceniamy jego wydajność za pomocą serii testów w symulowanym środowisku. W artykule omówiono implikacje tego podejścia, ze szczególnym uwzględnieniem unikania pętli i złożoności obliczeniowej, a także zasugerowano dalsze ulepszenia i potencjalne zastosowania.*

Słowa kluczowe: obliczenia rozproszone, skierowane grafy acykliczne, potokowe przetwarzanie danych

Introduction

Workflows, pipelines and systems enabling them are a mainstay of many different areas involving distributed computation and data processing. Primarily, the systems in question are based on Directed Acyclic Graphs, which in this domain represent activities and order-of-operations relationships between them. Thus, the graph of the workflow or a pipeline needs to be directed to maintain the order of operations, and acyclic to not cause an endless loop of computation.

In common workflow management systems the DAGs are explicit, rigid, developed and maintained by the operator of the system, which results in painstaking and meticulous development, maintenance and update process. An extreme scenario of that might be when a component commonly used in the pipelines receives a breaking change, rising a need to address the breaking change in every pipeline.

Thus, operator-defined DAG workflow runners, while gracefully solving tasks they are intended for, impose a non-insignificant burden, which should be considered and addressed when designing, maintaining and improving applications that rely on those systems.

This paper explores the possibility to execute workflows without actually defining and maintaining directed acyclic graphs by dynamically constructing and traversing a Directed Acyclic Graph from a set of defined Operators with defined input and output Data Kinds.

While exploring and constructing DAGs in an automated way is frequently done in various domains, like cumulative risk assessment [1] as well as epidemiology [4], the attention of these researches is mostly driven towards causal inference and evidence synthesis, which is an environment principally different from workflow construction and execution, since causal graphs are mainly a statistics tool.

1. Algorithm description

1.1. Conceptual overview

The main goal of the proposed algorithm is to enable a way of dynamically constructing and traversing a Directed Acyclic Graph using user defined data types and operations available on those, with the possibility for the operations to be non-deterministic in regards to their outputs, so that the operator could return all, some or none of its possible output types, and output them in any quantity desired. The traversal should be seeded by a set of input data marked with their corresponding types, and the algorithm should then be able to iterate over the data, and the provided operators until it exhausts all calculations possible with the input data, data returned by all executed operators and all available operators. The proposed algorithm should also be able to detect and avoid cycles so that the users of the envisioned system using this algorithm would be safeguarded against endless cyclic calculations and equipment and maintenance time required to prevent, detect and stop those.

To achieve the goals and requirements established, presented algorithm is designed around working with several primary entities, which together are used to give a complete description of a given execution with a set of given input, intermediary and output data:

- **Data Kind** is a semantic label that is used to annotate the required inputs of a given Operator, and any data that exists within the scope of a given execution. It should also be stated that the algorithm described here is not concerned with data validation, so Data Kinds are merely labels. One should conceptualize those as Types in Charles Sanders Peirce's Type-Token distinction.
- **Operator** is an atomic operation over a set of strictly defined inputs of various Data Kinds, and may or may not output a set

This work is licensed under a Creative Commons Attribution 4.0 International License.
Utwór dostępny jest na licencji Creative Commons Uznanie autorstwa 4.0 Międzynarodowe.

of outputs of arbitrary Kinds. Neither the size nor the Kind labels of the outputs of a given operator are deterministic from the perspective of the algorithm described here.

- **Data Unit** is a piece of data. If Data Kinds are Peirce's Types, then Data Units are Tokens. A given Data Unit consists of a Data Kind label for the item, a reference to a file containing said data, and an ancestry list, which describes the sequence of Operators which resulted in the existence of this Data Unit.
- **Step** is an execution of a given Operator with Data Units matched to the inputs of an Operator.

Evidently, this set of entities does not describe a Directed Acyclic Graph explicitly, instead opting for entities that are easy to describe and maintain for the user of the envisioned system exploiting proposed algorithm, thus the DAG exists from the combination of these entities implicitly, rather than explicitly, although it is possible to reconstruct it from these entities for visualization purposes.

1.2. Algorithm details

For convenience, the algorithm is split into several functions. In the following pseudocode, `O` is the set of all Operators in the system, `K` is the set of all Data Kinds, `S` is the set of all computed steps in a given execution, `U` is the set of all Data Units (input and output to the Operators) in a given execution.

Algorithm 1 describes the general order of operations of the proposed algorithm's environment. On each turn, we solve the next steps using the current Data Pool `U`, and execute those to get new Data Units to add to `U` for the next turn of computations.

It should also be mentioned that the main loop does not necessarily need to wait for all the computed steps to finish until solving the next turn of new steps. To cut out the wait time on longer operators, we suggest implementing this in a way that would compute the next steps every time at least one of the operators finishes executing. Since both the wait time optimization, and the pseudocode for the `ExecuteSteps` function are somewhat implementation specific and don't constitute the main proposal of this paper – we will omit them here.

Algorithm 1: Main loop

```
Data: O, K, U satisfying ∀u ∈ U, u.dataKind ∈ K containing input data provided by user
Result: U containing results of exhaustive computation with o ∈ O.
start:

S ← ∅;
repeat
    N ← SolveTurn(O,K,S,U);
    if N ≠ ∅ then
        U ← U ∪ ExecuteSteps(N,U);
        S ← S ∪ N;
    end
until N ≠ ∅
end

Function SolveTurn(O, K, S, U):
/* Get those operators, whose input kinds are satisfied with at least one unit in execution's data pool */
    O_I ← GetAvailableOperators(O, U);
    opsWithInputOptions ← map o ∈ O with
        SelectOperatorInputs(o, U);
    if opsWithInputOptions = ∅ then
        return ∅;
    end
/*  Per operator, make a cartesian product of all available units per input to create suitable sets of inputs that would constitute a Step to execute */
opsWithInputCombinations ← map
opsWithSatisfiedInputs with f(o) =
                (operator:o.operator, inputCombinations:
            {i_1…i_n ∈ o.unitsPerInput|i_1 × i_2 × … × i_n});
    newSteps ← ∅;
    foreach o ∈ opsWithInputCombinations do
        foreach i ∈ o.inputCombinations do
            if
                i does not contain duplicate units
then
                append (operator:o, input:i) to
                    availableNewSteps;
            end
        end
    end
    N ← {n ∈ newSteps | n ∉ S};
    return N
Function GetAvailableOperators(O,U):
    kindsPresent ← map u ∈ U to u.dataKind;
    return
            {o ∈ O| o.inputDataKinds ⊆ kindsPresent};
Function SelectOperatorInputs(o,U):
    matchingDataUnits ← filter U for u ∈ U,
        u.dataKind ∈ o.inputDataKinds;
    unitsPerInput ← empty Map;
    foreach input ∈ o.inputs do
        unitsPerInput[input.name] ←
                        empty List;
        foreach unit ∈ matchingDataUnits do
            if unit.dataKind = input.dataKind
                and ¬o.name ∈ unit.ancestors then
                append unit.id to
                    unitsPerInput[input.name];
            end
        end
    end
    return (operator, unitsPerInput);
```

Function `SolveTurn` in Algorithm 1 contains the main order of operations of proposed solution to the problem of discovery and traversal of an implied Directed Acyclic Graph. The turn is solved by first figuring out the operators that have the necessary inputs to be executed – so that each of the inputs has at least one Data Unit in the pool that matches the input's data kind. After that what we have is a set of data units per input satisfying operator's requirements. To produce a set of viable next steps to execute we need to transform the sets of units per operator into set of pairings, where the operator is paired with one data unit per input. Since we need to exhaust all possible ways to compute a given operator with given inputs, we accomplish that transformation with a cartesian product of N sets, where N is the amount of inputs of the operator. Once we have the steps we can attempt, the only thing left is to filter out those operator-units pairs which have not been yet executed.

By doing so, each time the Data Pool is appended with new Units we still retry all possible combinations. If we went the way of only checking the new Unit Kinds for possible operators, we would miss executing the operators that had part of their inputs satisfied with input data of the execution, part with (N-1)-th turn, and part with N-th turn.

2. Experimental setup

To test the robustness of this algorithm in a semi-realistic scenario, a test bench was devised. This test bench would randomly generate 50 data kinds and 25 operators. Operators are randomly generated to have 1-4 inputs. These inputs can be either from the starting or middle third of the Data Kind array, to facilitate formation of longer executions. The execution is then seeded with three Data Units. After each pass, a random amount of Data Units ranging from 0 to twice the amount of produced valid next steps would be added to the data pool as descendant from random steps to simulate random operator outputs.

The algorithm was implemented in TypeScript, and executed as part of an HTTP server on a GKE non-Autopilot cluster with one node. Time was measured using `performance.now()` precision timing API. This implementation showed adequate results – 0.5–1 ms for executions with <5 turns and <10 steps, 4–8 ms for executions with <5 turns and <50 executed steps

and 15–30 ms for executions with <15 turns and <50 executed steps. Time taken on this scale can be considered negligible when compared to time which would be spent by the system to spawn, prepare and execute Kubernetes jobs or other ways of executing operators.

3. Discussion

3.1. Cycle avoidance

Cycle avoidance in this approach is guaranteed by only suggesting units to operator inputs that do not have this operator in their ancestry. While in this basic approach it is suggested to only keep operators in the ancestry, it might be beneficial to track the ancestry unit-wise as well, which would enable the users of the system exploiting this algorithm to run an execution on a set of different datas of same type and still be able to track down which results were achieved from which input data, which while not good practice, is something that could be expected from real world use.

Another problem of cycle avoidance in this algorithm is the case of a hypothetical operator X taking units of kinds A and B as inputs, and providing A as the output. Currently, as illustrated by figure 1, if the operator X took A_1 and B_1, and produced A_2, and, by addition from the user or from some other operator, data unit B_2 would then appear, operator X would only be re-run with A_1 and B_2 as it's inputs, while in a theoretical scenario it might be beneficial to run X with A_2 and B_2 as well, despite A_2 being produced by operator X itself, as illustrated by figure 2.

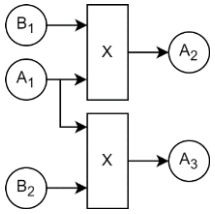


Fig. 1. Current solution to the hypothetical operator X scenario

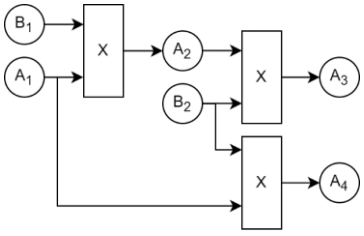


Fig. 2. Possible solution to the hypothetical operator X scenario

While it's assumed a described scenario is highly unlikely, it could be argued that a use case described in Fig. 2 is a more correct way to solve such a theoretical scenario.

3.2. Complexity

The complexity of presented algorithm is hard to point out precisely due to the nature of processing several entities at the same time and spawning new entities during the process, but we can still analyse some of the aspects of it.

The theoretical narrow point of presented algorithm is the computation of the cartesian product of sets of data units per operator input, which has a complexity of $O(2^n)$ with n being the number of operator inputs per specific operator. While $O(2^n)$ a horrible complexity, we consider that it's highly unlikely this will be a problem in the real world usage scenarios, since an operator with hundreds of inputs is quite hard to imagine and justify.

From the perspective of operator count (n) and data unit (m) count the complexity of this algorithm can be described as $O(nm)$.

3.3. Running at scale

While the pseudocode provided, and the preliminary implementation of this algorithm are designed and presented from a perspective of a central service which would orchestrate the computation, it is obvious that there are uses and scenarios where one service would not be enough to handle the amount of data flowing through it.

From our perspective, this algorithm can also be implemented in a decentralised and distributed manner by utilizing Pub/Sub [3] or other distributed communication model based tools to trigger both operator execution and the solution for the next possible steps. Given that operator executors and the solvers will be implemented in a stateless manner themselves, instead outsourcing state and data handling to other distributed systems, there should be no problems employing this approach in a decentralised manner.

3.4. Future work and potential applications

The presented algorithm is a basic attempt at the described problem. While it is a solution, there most probably are at least slightly better ways to compute an implied DAG in such a manner, from adding more features and possible nodes to the graph by implementing a more thought-through version of cycle avoidance, to optimizations in ways the operators, data units and possible next steps are filtered down to produce valid next steps not executed before.

The algorithm presented is intended to be used in software requiring automated workflow management systems as a replacement or a complement for CWL, Nextflow and other similar systems commonly used in data engineering pipelines and bioinformatical/biostatistical computations [5, 6], as well as for running data through neural networks [2]. While it is obvious this approach would not be a replacement for a significant majority of use-cases of workflow systems in general, the utilization of this algorithm in an envisioned system is predicted to enable easier development of prototypes and small-scale systems for scientific and bioinformatical computations relying on data pipelines with lots of common components by cutting down the time and effort spent on development and maintenance of pipelines. We see the presented approach as being especially valuable in circumstances with constantly changing requirements that grow from the new and new opportunities discovered during the process of developing with the suggested approach.

Designed around non-deterministic outputs, the presented algorithm also allows flexible and effortless branching and expansion in a way that is not possible with CWL. Figure 3 describes a use case for a workflow system employing presented algorithm to run three-dimensional CT scan analysis.

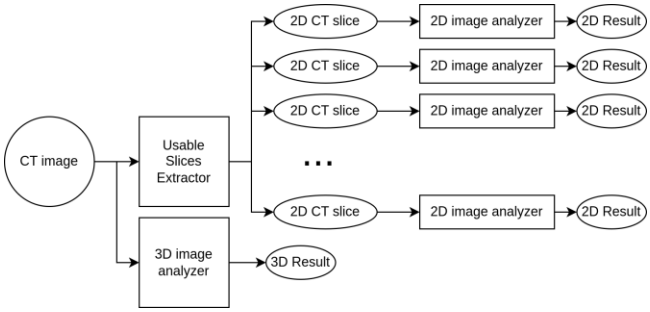


Fig. 3. Usage scenario for CT scan analysis

In this scenario, usable slice extractor would, for example, filter out the slices which contain the area of interest – for example, lungs in chest CT imaging, and pass them on to other analyzers. As it's impossible to predict how many slices would be actually containing the area of interest due to inherent variety in human bodies, the extractor would output an unpredictable

amount of Data Units, for each of which the system will automatically find and execute the according analysis operators. The system would also allow easy extensibility with additional three- and two-dimensional analyzers.

4. Conclusion

An algorithm for dynamic construction and traversal of a workflow DAG was presented, with an aim to reduce user effort in developing and maintaining workflows. The described algorithm iterates over data and operators in the system, selecting possible to run data-operator combinations on each step of the algorithm until there is no more data-operator pairings possible and not previously executed. The algorithm also avoids cyclic computations, although in a way which would not allow some theoretical uses. Experimental results indicate that the algorithm is robust and avoids stalling and endless computation and is able to generate and execute complex workflows, supporting operators with uncertain outputs. While the computation complexity of parts of this algorithm are not ideal, practical implications of it seem insignificant in real world situations. Future work will focus on refining cycle avoidance mechanisms, optimizing performance, and development of a Kubernetes-based system utilizing the presented algorithm.

Acknowledgements

Special thanks to Anton Shpigunov, Radek Janik, and Anton Zhdan-Pushkin for their valuable perspectives and insights during the development of the algorithm and the writing of this paper. Their contributions were instrumental in shaping both the technical and narrative aspects of this work.

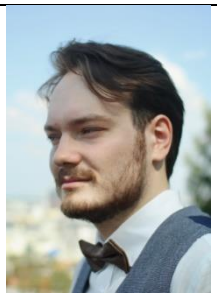
References

[1] Brewer L. E. et al.: Causal inference in cumulative risk assessment: The roles of directed acyclic graphs. Environment International 102, 2017, 30–41 [https://doi.org/https://doi.org/10.1016/j.envint.2016.12.005].
[2] Colonnelli I. et al.: Bringing AI pipelines onto cloud-HPC: setting a baseline for accuracy of COVID-19 diagnosis. ENEA CRESCO in the Fight Against COVID-19, 2021, 66–73 [https://doi.org/10.5281/ZENODO.5151511].
[3] Eugster P. Th. et al.: The many faces of publish/subscribe. ACM Comput. Surv. 35(2), 2003, 114–131.
[4] Ferguson K. D. et al.: Evidence synthesis for constructing directed acyclic graphs (ESC-DAGs): a novel and systematic method for building directed acyclic graphs. International Journal of Epidemiology 49(1), 2019, 322–329 [https://doi.org/10.1093/ije/dyz150].
[5] Georgeson P. et al.: Bionitio: demonstrating and facilitating best practices for bioinformatics command-line software. GigaScience 8(9), 2019, giz109 [https://doi.org/10.1093/gigascience/giz109].
[6] Jackson M. et al.: Using prototyping to choose a bioinformatics workflow management system. PLOS Computational Biology 17(2), 2021.

M.Sc. Fedir Smilianets
e-mail: fedor.smile@gmail.com

Fedir Smilianets received his M.Sc. from National Tecnhical University of Ukraine "Igor Sikorsky Polytechnic Institute". Since 2021, he is a Ph.D. student at the department of Computer Science and Software Engineering, also working as a teaching assistant there since 2023.
His research interests include machine learning and systems for data processing in the cloud.

https://orcid.org/0000-0002-0061-7479



Ph.D. Oleksii Finogenov
e-mail: fenyatrashbox@gmail.com

Ph.D., docent at Department of Computer Science and Software Engineering of National Technical University of Ukraine "Igor Sikorsky Polytechnic Institute".
Research interests include numerical analysis, optimization methods, parallel computing, decision making systems.

https://orcid.org/0000-0002-1708-5632