# Conference of PhD Students in Computer Science

*Guest Editor:*

**Attila Kertész**

Department of Software Engineering
University of Szeged
Szeged, Hungary
keratt@inf.u-szeged.hu

# Preface

The *11th Conference of PhD Students in Computer Science (CSCS)* was organized by the Institute of Informatics of the University of Szeged (SZTE) and held in Szeged, Hungary, between June 25–27, 2018.

The members of the *Scientific Committee* were the following representatives of the Hungarian doctoral schools in computer science: János Csirik (Co-Chair, SZTE), Lajos Rónyai (Co-Chair, SZTAKI, BME), Péter Baranyi (SZE), András Benczúr (ELTE), András Benczúr (SZTAKI), Hassan Charaf (BME), Tibor Csendes (SZTE), László Cser (BCE), Erzsébet Csuhaj-Varjú (ELTE), József Dombi (SZTE), István Fazekas (DE), Zoltán Fülöp (SZTE), Aurél Galántai (ÓE), Zoltán Gingl (SZTE), Tibor Gyimóthy (SZTE), Katalin Hangos (PE), Zoltán Horváth (ELTE), Márk Jelasity (SZTE), Zoltán Kása (Sapientia EMTE), László Kóczy (SZE), János Levendovszki (BME), Gyöngyvér Márton (Sapientia EMTE), Branko Milosavljevic (UNS), Valerie Novitzka (TUKE), László Nyúl (SZTE), Marius Otesteanu (UPT), Attila Pethő (DE), Vlado Stankovski (UNILJ), Tamás Szirányi (SZTAKI), Péter Szolgay (PPKE), János Sztrik (DE), János Tapolcai (BME), János Végh (ME), and Daniela Zaharie (UVT). The members of the *Organizing Committee* were: Attila Kertész (Chair), Balázs Bánhelyi, Tamás Gergely, and Zoltán Kincses.

There were more than 55 participants and 52 talks in several fields of computer science and its applications (13 sessions). The talks were going in sections in Artificial Intelligence, Static Analysis, Cloud Computing I., Testing, Cloud Computing II., Image Processing I., Education, Image Processing II., Optimization, Algorithms, Programming Languages, Evaluation, Business Process. The talks of the students were completed by 3 plenary talks of leading scientists: Bálint Daróczy (MTA SZTAKI, Hungary), Michael C. Mackey (McGill University, Canada), and Massimiliano Di Penta (University of Sannio, Italy).

The open-access scientific journal Acta Cybernetica offered PhD students to publish the paper version of their presentations after a careful selection and review process. Altogether 24 manuscripts were submitted for review, out of which 10 were accepted for publication in the present special issue of Acta Cybernetica. 2 papers were published in the previous issue, and 2 additional papers are planned to be published in a future issue.

The full program of the conference, the collection of the abstracts and further information can be found at `http://www.inf.u-szeged.hu/~cscs`.

On the basis of our repeated positive experiences, the conference will be organized in the future, too. According to the present plans, the next meeting will be held around the end of June 2020 in Szeged.

*Attila Kertész*
Guest Editor

Acta Cybernetica **24** (2019) 5–16.

# Towards a Classification-Based Systematic Approach to Facilitate the Design of Domain-Specific Visual Languages*

Sándor Bácsi[a] and Gergely Mezei[b]

**Abstract**

Domain-specific visual languages (DSVLs) are specialized modeling languages that allow the effective management of the behavior and the structure of software programs and systems in a specific domain. Each DSVL has its specific structural and graphical characteristics depending on the problem domain. In the recent decade, a wide range of tools and methodologies have been introduced to support the design of DSVLs for various domains, therefore it can be a challenging task to choose the most appropriate technique for the design process. Our research aims to present a classification-based systematic approach to guide the identification of the most relevant and appropriate methodologies in the given scenario. The approach can be capable enough to provide a clear and precise understanding of the main aspects that can facilitate the design of DSVLs.

**Keywords:** domain-specific visual languages, modeling, classification

## 1 Introduction

In software development, there has always been a big demand for improving the productivity and the speed of the development process by increasing abstraction. This is the main reason why model-driven software development [1] has become a promising paradigm among software developers and researchers in the past decades. Software models are mainly used for designing complex structures or systems in order to be able to specify the requirements on a higher abstraction level. Thus, the model can give a better overview of the system and help to understand the concepts of the targeted domain.

*The research has been supported by the European Union, co-financed by the European Social Fund (EFOP-3.6.2-16-2017-00013, Thematic Fundamental Research Collaborations Grounding Innovation in Informatics and Infocommunications).

[a]Department of Automation and Applied Informatics, Budapest University of Technology and Economics E-mail: bacsi.sandor93@gmail.com

[b]Department of Automation and Applied Informatics, Budapest University of Technology and Economics E-mail: gmezei@aut.bme.hu

DOI: 10.14232/actacyb.24.1.2019.2

Domain-specific languages (DSLs) [10] are specialized modeling languages that can efficiently raise the level of abstraction by using the concepts and the characteristics of the specific problem domain. DSLs are in a contrast to general-purpose languages like C, Python or Haskell that are designed to let developers write any sort of program with any sort of logic broadly applicable across domains. DSLs allow the effective management of the behavior and the structure of software programs and systems. Domain-specific visual languages (DSVLs) [15], compared to textual DSLs, can further improve the expressiveness and the usability of the given model. As a well-designed DSVL raises the level of abstraction, it also helps in hiding irrelevant, technical details and in emphasizing the domain-related parts in the models. In order to achieve this, it is essential to find the best visualization in design-time in order to satisfy the needs of the targeted domain.

There are several advantages of using a DSVL. The richness of the visual representation can simplify the modeling process and increase flexibility, thus DSVLs can be intuitively usable. As most of the people tend to associate a visualization for their problems, visual models can facilitate to understand the concepts and the main relations in the targeted domain. It can be easier to explain the main characteristics of a domain problem by using visual notations.

Compared to textual languages, DSVLs may have their drawbacks. DSVLs can be restrictive, since they may limit the freedom of creating complex language constructs. The visual entities representing a complex code can be hard or impossible to grasp in one glance. It can be challenging to find the effective visual way of expressing some advanced concepts, such as type systems, that can be found in most of the general-purpose textual programming languages. If DSVL is badly designed and it is used in a particular situation, the advantages may easily turn into disadvantages, thus it is essential to avoid counter-productive decisions by choosing the most appropriate representational concepts in design-time. A guideline can help in providing a clear and precise understanding of the main aspects to design the most suitable DSVL.

The high level of customization possibilities has its price: unlike in UML, each problem domain requires a custom, different visual representation to meet the requirements of the targeted domain. The exactness of the choice depends on how expressively the chosen concepts describe the DSVL and on the specific needs of the targeted domain.

In this paper, we present the main results of our classification methodology for visual domain-specific languages. We analyzed a wide range of existing DSVL methodologies and also created several illustrating examples for different domains to exemplify the most relevant graphical and structural characteristics. We used two metamodeling frameworks (Eclipse Modelling Framework [6] and Visual Modeling and Transformation System [18]) and a visual programming editor builder (Google Blockly [2], [7], [12]) to examine the most applicable methodologies.

The paper is organized as follows: Section 2 presents the background and the related work. Section 3 introduces our approach in order to give an understanding of the main concepts. Section 4 presents some of the illustrating examples which we elaborated, while concluding remarks are outlined in Section 5.

## 2 Related work

Various kinds of classifications have been created in the past to support the design process of DSVLs. However, most of these approaches are quite old, there is no relevant publication in the field for more than ten years. Due to the increasing use of DSVLs, a wide range of new tools and methodologies have been introduced recently based on completely new ideas. Our research aims to analyze and compare the most relevant methodologies on a larger scope which can support the design of new domain-specific visual languages with the new technologies. Different classifications of DSVLs have been presented in the literature. Basically, these classifications serve a completely different purpose than the one introduced in this research.

The principles in [9] are aimed at creating a hierarchy for visual languages which is based on the constraint multiset grammar formalism. The approach also takes into consideration the expressiveness and the cost of parsing for different classes. This approach is mainly based on formalism, rather than on the pragmatic use of DSVLs.

Myers [11] discusses programming systems and it is divided into categories using the orthogonal criteria of being visual programming or not, example-based programming or not, and interpretive or compiled. Similarly, in another paper [5] the authors presented a classification system, in which visual languages are categorized based on the visual programming paradigm they express and different visual representations.

There is another classification approach [3] which presents a suite of metamodels as a basis for a classification of visual languages. This approach introduces general metamodel patterns which can serve as a basis for different aspects that can facilitate the design of DSVLs. However, the approach does not take into consideration the possible non-metamodeling concepts and the pragmatic use of DSVLs.

There is a wide range of existing professional general-purpose modeling languages in the field of software engineering. For example, UML [17] and SysML [16] are intended to provide a standard way to visualize the design of different systems. Here, it is important to emphasize that our research focuses on creating new domain-specific visual languages considering the requirements of a certain domain, thus universal, standardized visual languages are not taken into account.

Our classification-based approach is not intended to be superior to other classification-based methodologies, it serves supplementary purposes. Our approach is mainly based on the nature of the graphical objects that compose the visual language, the connection types among the graphical objects, the composition rules and the visual representations. We also consider non-metamodeling approaches and compare them to metamodeling methodologies. In this way, we can provide a clear and precise understanding of the main aspects that can facilitate the design of DSVLs. We introduce a step-by-step guide on how to use our systematic approach in different design scenarios.

# 3 Classification-Based Systematic Approach

In this section, we present the steps of our classification-based systematic approach. Each subsection represents a step of our methodology. It is important to emphasize that not all of the steps can be used directly in all possible design scenarios. Some of the steps ( Section 3.1, Section 3.2 and 3.3) are meant to decide between a couple of mutually exclusive choices, while others (Section 3.4 and Section 3.5) are used only as a supporting step helping to fine-tune previous decisions.

## 3.1 Step 1: Flow type

Based on the flow type, domain-specific visual languages can be grouped into three subclasses: data flow languages, control flow languages and languages with no flow.

Data flow languages visualize the steps of data processing. Data flow concepts are based on the idea of disconnecting computational actors into stages that can execute concurrently. Data flow DSVLs visualize the processes that are undertaken, the data produced and consumed by each process, and the accumulative graphical objects needed to hold the data. It is possible to visualize what the system will accomplish by the flow of data.

Control flow visual languages visualize the logic of computation by describing its control flow. Control flow DSVLs graphically express the order in which instructions or statements are executed or evaluated. The graphical objects mainly represent the control structures and conditional expressions of the language, thus it is possible to visualize how the system will operate by the flow of control.

There are DSVLs which are neither data flow nor control flow because they target a static domain problem. These languages are used mainly to represent the structure of a system or a program, therefore no flow has to be described. A widely used example of no-flow graphical modeling languages is the UML class diagram, in which the structure of the system is described by the classes and the connections among them.

## 3.2 Step 2: Relation type

Based on the relation type, domain-specific visual languages can be grouped into two subclasses: containment-based and connection-based subclasses.

In the case of containment-based languages, entities are limited to embed in each other to express sentences of the targeted domain, no other types of connections (e.g. association, or inheritance) are supported. As the customization of embedding, graphical entities may be attached to other entities (e.g. representing methods and their parameters) and chained together (as in a call stack). To support this behavior, a predefined set of containment rules or constraints have to be specified to restrict the way of embedding, attaching and chaining. Blockly and Scratch [13] are widely used examples of containment-based languages, both support building blocks that can be connected like puzzle pieces in order to create easy-to-understand visual sentences.

Connection-based languages consist of two different kinds of building elements: entities and connections, i.e. nodes and edges. While data is usually expressed by entities, the flow of the model and the relations among entities are defined by connections. Connections may also have properties to ensure the customization of the relations among entities. Moreover, connections may also interpret containment as the container and the contained elements can be connected by a specialized containment-typed edge. This means that this category is more general, however its complexity is not needed in many practical cases.

## 3.3 Step 3: Methods of the abstract syntax definition

There are two key methods for the abstract syntax definition of a DSVL: metamodeling-, and non-metamodeling approaches.

Metamodeling methodologies provide methods for defining DSVLs based on the abstract notion of visual entities and of relations among them. These frameworks are capable of specifying the abstract syntax of a DSVL and expressing the additional semantics of existing information. The metamodel can expressively define the structure, semantics, and constraints for a family of graphical models. On the other hand, when a metamodel is instantiated, its elements become types, which can be instantiated in the instance models. Hence, complex structures and relations can be described in a flexible way by the usage of metamodeling concepts.

While metamodeling methodologies are based on various kinds of instantiation techniques, non-metamodeling approaches provide a somewhat simpler, template-based structure for creating visual entities. The main characteristic of non-metamodeling approaches is that they have a limited set of features which can be used on the different abstraction levels, thus complex structures cannot be visualized flexibly and expressively. One of the newest non-metamodeling approaches is Blockly. It supports a large set of features for different domains. In Blockly, the graphical objects are called blocks which can be customized as the basic building elements of the language. However, due to the template-based and weakly typed structure, complex type constraints cannot be applied.

## 3.4 Step 4: The way of the problem description

This is a fine-tuning step, since this step rather depends on the specific nature of the problem domain and also on the needs and preferences of the users. Based on the way of the problem description, domain-specific visual languages can be grouped into two subclasses: imperative and declarative languages.

Declarative visual languages describe the logic of computation. For example, SparqlBlocks [4] is a declarative DSVL developed in Blockly. Declarative languages visualize sets of declarations or declarative statements. Each of these visual declarations has a meaning depending on the targeted domain and may be understood independently. A declarative style of visualization helps to understand the problems of the targeted domain and the approach that the system takes towards the

solution of the problem, but is less expressive on the matter of mechanics which describe the flow of the system.

Imperative visual languages consist of visual statements that change the state of a program or a system. For example, Scratch is an imperative visual programming language. The visualized statements express the way of execution of which results in a decision being made as to which of two or more visualized paths to follow. In imperative languages, the visual sentences can be created by sequences of commands, each of which performs some action. These actions may or may not have a dedicated meaning in the targeted problem domain.

### 3.5 Step 5: Visual representation

This is also a fine-tuning step, since it is related the concrete syntax of the language and it strongly depends on the needs and preferences of the users. DSVLs have a visual concrete syntax used for the representation of graphical elements and connections. Based on the visual representation, there are two key design aspects: iconic and diagrammatic visual representation.

In the case of iconic languages, entities are visualized by icons. For example, Lego Wedo 2.0 Software [8] provides an iconic visual language for educational purposes. The iconic language is a structured set of related icons. An icon can be attached to or composed of other icons, thus expressing a more complex visual concept.

Diagrammatic languages are mainly composed of elements with a pre-defined symbolic representation of information. The building blocks of diagrammatic languages such as geometric shapes are often connected by lines, arrows, or other visual links. Chart-like, schematic-like and graph-based visual languages are the most widely used examples.

The most important difference between iconic and diagrammatic languages is that icons are pre-defined and they have limited flexibility, while graphical building blocks of diagrammatic languages can be calculated and customized freely.

## 4 Illustrating examples

We can investigate some advantages and disadvantages of different approaches by solving various domain problems. In this section, we introduce two illustrating examples to present different design scenarios built upon the classification-based approach presented. Through the examples we only investigate the mutually exclusive steps from Step 1 to Step 3 because they specify the structural characteristics of the DSVL.

### 4.1 Logic gates

The first illustrating example demonstrates the domain of logic gates. In this domain, logic gates can perform logical operations on one or more binary inputs

and produce a single binary output. For the sake of simplicity, we can use AND, OR and NOT gates. There are visual entities which can only transmit a binary signal, while other visual entities can only receive the signals, therefore it is possible to create entities with a single input or output.

**Step 1**: We have to make our first design decision based upon the first step of the systematic approach. It is certain that we have to design a control flow language, because logic gates can be cascaded in the same way that Boolean functions can be composed, allowing the construction and transmission of all of Boolean logic, and therefore, all of the mathematics and algorithms that can be described.

**Step 2**: The next question is whether the DSVL fits the connection-based or the containment-based approach. The answer is not that simple as it seems at first glance. If we choose the connection-based option, logic gates can be represented as nodes that can be connected with edges, creating thus a connection-based language. Node-like model elements can be connected to each other, where we use ports instead to define the interface of a node. For example, a logic gate OR can have two input ports for the operands and a single output port, for its result. If we choose the containment-based option, it is very difficult to express the connection among logic gates, since no edges can be used. On the other hand, it can be hard to customize the interface of logic gates. In conclusion, the DSVL fits better the connection-based approach.

**Step 3**: In this step, we have to make our decision regarding the abstract-syntax definition. It is clear that complex structures and relations can be described in a flexible way by the usage of metamodeling concepts. Taken the previous structural decisions into account, it can be more effective to use a metamodeling methodology. We used VMTS to define the abstract syntax of the language and to set up custom visualization for the graphical editing. To support this behavior, VMTS allows to define the so-called meta ports on nodes. Figure 1 shows a half adder model as an example in VMTS. Here, it is important to emphasize that due to the connection-based nature the output result of the given node can be used for more inputs, therefore particular nodes with the same logic do not have to be duplicated.

**Alternative solution**: For the sake of completeness, we tried a different design scenario in our classification-based approach to prove the importance of the appropriate design decisions. Let us assume the following design scenario: Unlike the previous design scenario, after Step 1, we can make a different decision. In Step 2 we choose the containment-based approach even if we are aware of the fact that this is not the better option. In Step 3 we decide to use a non-metamodeling approach. We used Blockly to create the containment-based variant of this example. While it is easy to define the blocks themselves, it is very difficult to express the connection among logic gates, since no edges can be used. Blocks have to be duplicated and there can only be one output on a block - the left output. Figure 2 shows the same model as in the connection-based example, but visualized in Blockly. In this example, the AND logic has to be used twice from input A and B to be able to implement the half adder logic. Even the input A and B visual entities have to be duplicated.

As the illustrating example shows, dealing with multiple connections in a control

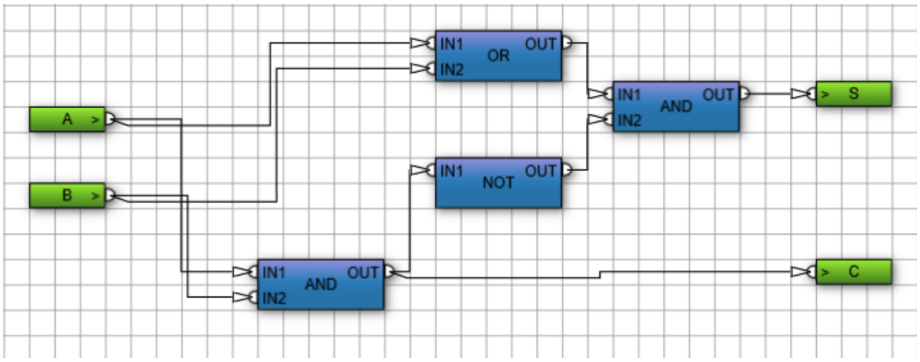



Figure 1: Half adder example in VMTS

flow domain may have its drawbacks in the containment-based approach. It is more expressive to use the principles of the connection-based approach to graphically express the order in which instructions or statements are executed or evaluated.

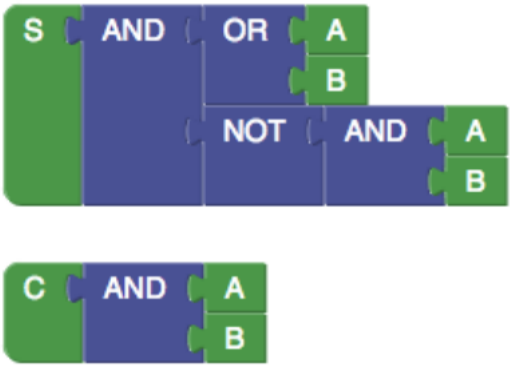



Figure 2: Half adder example in Blockly

## 4.2 Departments of a company

In this illustrating example, we present a simple DSVL for modeling the departments of a company. Let us assume the following specification: *A company has different departments. Employees work in departments. Employees may have a principal and every department has exactly one director.*

**Step 1**: At first, we have to make our first design decisions to identify the flow type of the domain. It is certain that we have to design a no-flow language, because no flow has to be described, only the static relations among entities are to be modeled.

**Step 2**: The next question is whether the DSVL will be connection-based or containment-based. If we choose the connection-based option we can definitely visualize the employee - principal relationship with some kind of visual links. On the other hand, it can be difficult to visualize the employee - department relation, because after a certain amount of employees the visual entities can be hard or impossible to grasp in one glance. In conclusion, besides the connection-based approach it would be advantageous to use additional containment-based nature for the employee - department relation.

**Step 3**: It can be easier to express the aforementioned structural characteristics by using a metamodeling methodology. We used EMF to create the connection-based variant of the DSVL. EMF provides effective features to express the basic relations among entities in order to define the abstract syntax of the DSVL. Based on EMF, Sirius [14] provides useful features for the customization of concrete syntax. Figure 3 shows a visualized model as an example. In this simple demonstration,

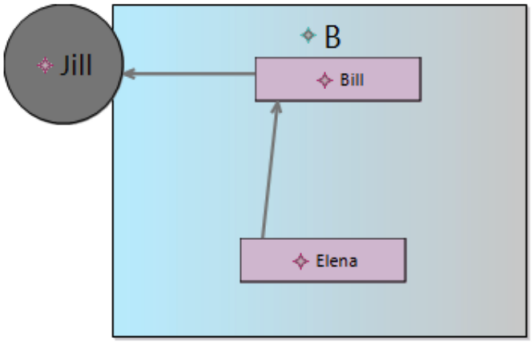



Figure 3: Departments example in Sirius

we used a rectangular box notation for the departments. Arrow notations are used to express the employee-principal relationship and a circle notation is used to visualize the head of the given department. Beside the connection-based patterns, this example has containment-based nature since employee notations can be embedded in departments. Here, it is worth to emphasize that due to the connection-based structure no entity has to be duplicated visually, because they can be connected with the arrow notations to express the employee-principal relationship.

**Alternative solution**: As the second solution, we elaborated a different design scenario. After Step 1, we can make a different decision. In Step 2 we choose the pure containment-based approach even if we know that it will be hard to express every relation by using just only the principles of the containment-based approach. In Step 3 we decide to use a non-metamodeling approach, Blockly to create the pure containment-based variant of this illustrating example. We used a container block to express the department relationship. The head of the department can be

connected to the *department block*. *Person-principal blocks* can be embedded into the department block to express which employees work in the given department. *Person-principal* blocks can express the hierarchical relationship among employees and principals, however it is more inconvenient and less expressive than in the connection-based approach because blocks have to be duplicated. Figure 4 shows the same model as in the previous example visualized by the principles of the containment-based approach.

In general, for a no-flow domain it is not recommended to use exclusively the concepts of the containment-based approach. On the other hand, in some cases it can be advantageous to let embedding of visual entities even for connection-based languages.

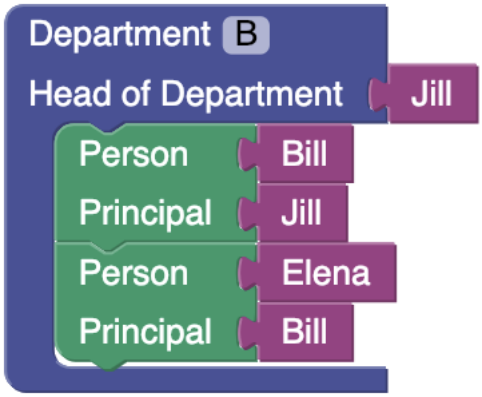



Figure 4: Departments example in Blockly

## 5 Conclusions

In this paper, we presented several aspects of the classification-based systematic approach for domain-specific visual languages. We believe that the approach can be used as a guide while designing DSVLs. With the help of these guidelines it is now easier to analyze the characteristics of the language and to associate it to an appropriate solution.

We also analyzed the features of Eclipse Modeling Framework, VMTS and Blockly based on different illustrating examples that we created for our classification methodology. We realized that due to the limitations of Blockly, many complex problems cannot be described expressively because aggregations, references and composition rules are missing from its developer framework. Despite the limitations of Blockly, it provides a flexible and easy way to learn to design DSVLs based on containment-based aspects. Unlike Blockly, both EMF and VMTS provide a large feature set for the abstract syntax definition, but they are not as effective and intuitive as Blockly in the definition of containment-based languages.

Further investigations are necessary to validate the kinds of conclusions that can be drawn from this paper. In the future, we aim to create a framework to support the design of visual-domain specific languages based on a questionnaire built upon the methodology presented. It would be beneficial to capture a description of a DSVL from an end-user perspective and give recommendation based on the specification and the specific needs of the targeted domain. The framework should also support an intuitively usable way of designing DSVLs even for complex language constructs and it could assist to align the design of DSVLs to best practices and also benchmark and analyze different design processes. Further studies should investigate how to consider the extensions of existing languages (e.g UML profiles) in the context of our methodology. Therefore, we are also working on new illustrative examples and analyzing other existing approaches to create a more detailed classification-based systematic approach.

# References

[1] Beydeda, Sami, Book, Matthias, Gruhn, Volker, et al. *Model-driven software development*, volume 15. Springer, 2005.

[2] Blockly website. https://developers.google.com/blockly/. Accessed: 2018-08-22.

[3] Bottoni, P. and Grau, A. A suite of metamodels as a basis for a classification of visual languages. In *2004 IEEE Symposium on Visual Languages - Human Centric Computing*, pages 83–90, Sept 2004. DOI: 10.1109/VLHCC.2004.5.

[4] Bottoni, Paolo and Ceriani, Miguel. Sparql playground: A block programming tool to experiment with sparql. In *VOILA@ISWC*, 2015.

[5] Burnett, Margaret M. and Baker, Marla J. A classification system for visual programming languages. *J. Vis. Lang. Comput.*, 5:287–300, 1994.

[6] Emf website. http://www.eclipse.org/modeling/emf/. Accessed: 2018-08-25.

[7] Fraser, Neil. Ten things we've learned from blockly. In *Blocks and Beyond Workshop (Blocks and Beyond), 2015 IEEE*, pages 49–50. IEEE, 2015.

[8] Lego wedo 2.0. https://education.lego.com/en-us/downloads/wedo-2/software. Accessed: 2018-08-21.

[9] Marriott, Kim and Meyer, Bernd. On the classification of visual languages by grammar hierarchies. *Journal of Visual Languages and Computing*, pages 375 – 402, 1997.

[10] Mernik, Marjan, Heering, Jan, and Sloane, Anthony M. When and how to develop domain-specific languages. *ACM Comput. Surv.*, 37(4):316–344, December 2005. DOI: 10.1145/1118890.1118892.

[11] Myers, Brad A. Taxonomies of visual programming and program visualization. *J. Vis. Lang. Comput.*, 1(1):97–123, March 1990. DOI: 10.1016/S1045-926X(05)80036-9.

[12] Pasternak, Erik, Fenichel, Rachel, and Marshall, Andrew N. Tips for creating a block language with blockly. In *Blocks and Beyond Workshop (B&B), 2017 IEEE*, pages 21–24. IEEE, 2017.

[13] Scratch. https://scratch.mit.edu/. Accessed: 2018-08-25.

[14] Sirius website. https://www.eclipse.org/sirius/. Accessed: 2018-08-20.

[15] Sprinkle, Jonathan and Karsai, Gabor. A domain-specific visual language for domain model evolution. *Journal of Visual Languages & Computing*, 15(3-4):291–307, 2004.

[16] Sysml. https://sysml.org/. Accessed: 2018-08-29.

[17] Uml. http://www.uml.org/. Accessed: 2018-08-29.

[18] Vmts website. www.aut.bme.hu/Pages/Research/VMTS/Introduction. Accessed: 2018-08-21.

Acta Cybernetica **24** (2019) 17–28.

# Operations on Signed Distance Functions[a]

Csaba Bálint[b], Gábor Valasek[b], and Lajos Gergó[b]

**Abstract**

We present a theoretical overview of signed distance functions and analyze how this representation changes when applying an offset transformation. First, we analyze the properties of signed distance and the sets they describe.

Second, we introduce our main theorem regarding the distance to an offset set in $(X, ||\cdot||)$ strictly normed Banach spaces. An offset set of $D \subseteq X$ is the set of points equidistant to $D$. We show when such a set can be represented by $f(\boldsymbol{x}) - c = 0$, where $c \neq 0$ denotes the radius of the offset. Finally, we apply these results to gain a deeper insight into offsetting surfaces defined by signed distance functions.

**Keywords:** signed distance functions, sphere tracing, computer graphics

## 1 Introduction

Surface representations for real-time graphics rely on linear approximations. With the advent of hardware accelerated tessellation units, parametric surfaces gained momentum in real-time computer graphics; however, implicit mappings are still considered infeasible for high-performance applications [2, 4, 6, 7, 9, 13].

Nevertheless, implicit functions simplify some otherwise challenging operations. For example, blending between different shapes does not necessitate the explicit representation of the target topologies when both objects are represented implicitly [3, 19]. Similarly, the result of set operations on these objects can be trivially computed [7, 12, 14, 15].

Our paper focuses on a particular class of implicit representations, signed distance functions (SDFs). Hart noted in [10] that SDFs could be rendered efficiently using a technique called sphere tracing [2, 9, 16]. This algorithm and the constant evolution of GPUs opened up the possibility of incorporating implicit representations into real-time applications, as exemplified by [1, 6, 18] more recently.

We discuss this class of functions and highlight their theoretical aspects that have practical consequences in rendering. In particular, we focus on offsetting SDF representations. Although both offsets and SDFs are simple concepts, their

[a]The project has been supported by the European Union, co-financed by the European Social Fund (EFOP-3.6.3-VEKOP-16-2017-00001).

[b]Eötvös Loránd University, E-mail: {csabix,valasek,gergo}@inf.elte.hu

DOI: 10.14232/actacyb.24.1.2019.3

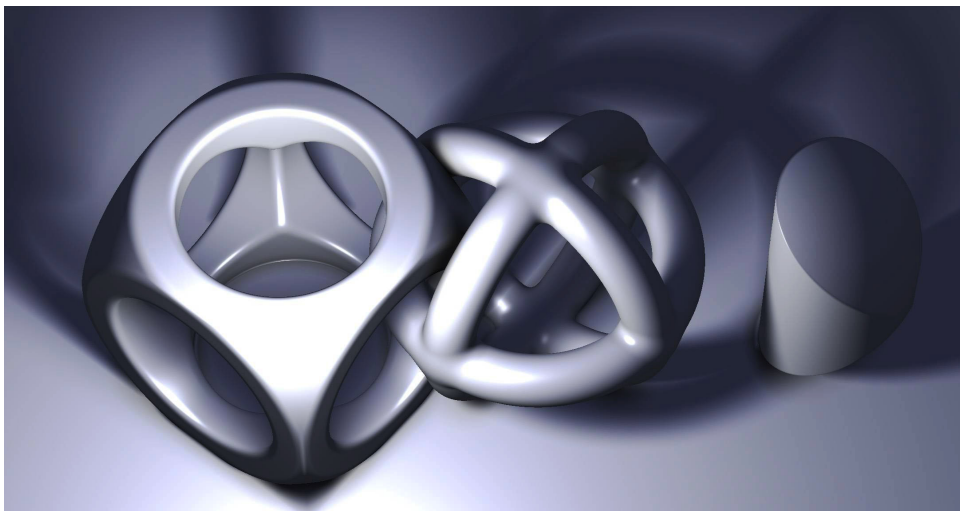


Figure 1: A scene modeled with and rendered using signed distance functions.

combination does not always yield the expected simplicity when one tries to find a representation for the result, as highlighted in Section 6. Our paper begins with a set-theoretic overview in Section 2. We base our theorems upon these results.

Section 3 present a general algorithm for displaying surfaces defined by implicit functions, whereas Section 4 demonstrates the power SDFs provide in speeding up such tasks and their practical importance.

In Section 5, we propose a slightly different definition for signed distance function than seen in [10]. We show that the two definitions are equivalent.

We present our main result in Section 6. We show that it is possible to represent the radius $c \neq 0$ offset of $f(\boldsymbol{x}) = 0$ by $f(\boldsymbol{x}) - c = 0$; however, $f - c$ only produces a signed distance function on the subset of $\mathbb{R}^3$ for which $\frac{f(\boldsymbol{x})}{c} \geq 1$.

It has been observed that adding a constant value to a signed distance function produces a function that defines the offset set of the original surface [8, 10, 17]. In this paper, we analyze this operation mathematically and explain the reasons behind the effectiveness and limitations of the practical solutions.

## 2 Set-theoretic basics

This section reviews the definitions and results from the literature our paper relies on. Dyer et al. explain the topic in more detail in [5]. Let $(X, d)$ denote a metric space. We also use $d : X \times X \to [0, +\infty]$ to denote the distance to a set.

**Definition 1** (Distance to set)**.** *Let $A \subseteq X, \boldsymbol{p} \in X$. Then*

$$d(\boldsymbol{p}, A) := \inf_{\boldsymbol{a} \in A} d(\boldsymbol{p}, \boldsymbol{a})$$

*denotes the distance of* $\boldsymbol{p}$ *from the set* $A$*. Let* $\inf \emptyset := +\infty$.

**Definition 2** (Neighborhood)**.** *Let us denote the* $r > 0$ *radius neighborhood of an element* $\boldsymbol{p} \in X$ *by*

$$\mathcal{S}_r(\boldsymbol{p}) := \{\boldsymbol{x} \in X \ : \ d(\boldsymbol{x}, \boldsymbol{p}) < r\} \ .$$

$A \subseteq X$ is open if $\forall\, \boldsymbol{a} \in A$, $\exists\, \epsilon > 0 \ : \ \mathcal{S}_\epsilon(\boldsymbol{a}) \subseteq A$. The set $B \subseteq X$ is closed if $X \setminus B$ is open. Note that $\emptyset$ and $X$ are both closed and open.

$C \subseteq X$ is compact if every open covering of it can be reduced to be of finite cardinality. A compact set is closed and bounded, i.e. $\exists\, R > 0$ such that $C \subseteq \mathcal{S}_R(\boldsymbol{0})$. A bounded and closed set is compact if $X$ is a finite dimensional metric space, for example $X = \mathbb{R}^3$.

**Lemma 1** (Existence of extremal element)**.** *Suppose* $A \subseteq X$ *is closed and* $\boldsymbol{x} \in X$ *where* $(X, d)$ *is a complete metric space. Then*

$$\exists\, \boldsymbol{a} \in A \ : \ d(\boldsymbol{x}, A) = d(\boldsymbol{x}, \boldsymbol{a})$$

The proof for Lemma 1 can be found in [11] on page 102 for $\mathbb{R}^n$, the proof is analogous for this case [11, 5].

Furthermore, we denote the interior of the set $A \subseteq X$ as

$$\operatorname{int} A := \{\boldsymbol{a} \in A \mid \exists\, \epsilon > 0 \ : \ \mathcal{S}_\epsilon(\boldsymbol{a}) \subseteq A \ \}$$

The closure of $A \subseteq X$ is

$$\overline{A} := \{\boldsymbol{a} \in X \mid \forall\, \epsilon > 0 \ : \ \mathcal{S}_\epsilon(\boldsymbol{a}) \cap A \neq \emptyset\}$$

The boundary of $A$ is denoted by $\partial A := \overline{A} \setminus \operatorname{int} A$. For any set $A \subseteq X$ it follows from the definitions that $\operatorname{int} A$ is open, $\overline{A}$ and $\partial A$ are closed sets.

# 3 Raymarching

From now on, let us consider surfaces defined by an $f : \mathbb{R}^3 \to \mathbb{R}$ implicit function, such that the surface is the $\{f \equiv 0\} := \{\boldsymbol{x} \in \mathbb{R}^3 \mid f(\boldsymbol{x}) = 0\}$ level-set. For example, the characteristic function $1 - \mathcal{X}_D = \mathcal{X}_{\mathbb{R}^3 \setminus D} : \mathbb{R}^3 \to \{0, 1\}$ is an implicit function of any $D \subseteq \mathbb{R}^3$ set.

A ray is a half line originating from a particular point, for example, the camera. Let us represent rays by their origin $\boldsymbol{p} \in \mathbb{R}^3$ and unit length direction vector $\boldsymbol{v} \in \mathbb{R}^3$, $\|\boldsymbol{v}\|_2 = 1$. Then a ray is written as

$$\boldsymbol{s}(t) := \boldsymbol{s}_{\boldsymbol{p},\boldsymbol{v}}(t) := \boldsymbol{p} + t \cdot \boldsymbol{v} \in \mathbb{R}^3 \quad (t \geq 0) \ .$$

Therefore, the ray-surface intersection problem can be expressed as a root finding problem. We need to find the smallest positive root of the

$$f \circ \boldsymbol{s} \ : \ [0, +\infty) \to \mathbb{R}$$

**Algorithm 1** Raymarching a continuous implicit surface

**Input:** Ray defined by $\boldsymbol{p}$ and $\boldsymbol{v} \in \mathbb{R}^3$, where $\|\boldsymbol{v}\|_2 = 1$

**Input:** Continuous implicit function $f : \mathbb{R}^3 \to \mathbb{R}$

**Input:** $\Delta t > 0$ step size

**Output:** $t \in [0, +\infty)$ distance traveled along the ray

```
1: t := 0;   f_0 := f(s(0));   f_1 := f(s(Δt))
2: for t < t_max and f_0 · f_1 > 0 do
3:     t := t + Δt;                          Raymarch cycle – the bottleneck
4:     f_0 := f_1;
5:     f_1 := f(s(t));
6: end for
7: t := RefineSolution(f ∘ s, [t−Δt, t]);    For example, using secant method
8: return t
```

composite function. Usually, one can infer that $f$ is continuous in which case raymarching that is shown in Algorithm 1 can be used to find an approximate solution. The method takes $\Delta t$ sized steps along the ray looking for two consecutive values of different signs.

Despite being a popular algorithm for implicit surface rendering, raymarching is expensive, and it may even skip over solutions, causing visible artifacts. To provide a better ray tracing algorithm, $f$ needs to be restricted even further which is explained in the next section.

# 4 Sphere Tracing

Throughout this section, we adapt the definitions from Hart [10]. Let us consider the Banach-space $(\mathbb{R}^3, \|.\|_2)$ where we denote the induced metric as $d(\boldsymbol{x}, \boldsymbol{y}) := \|\boldsymbol{y} - \boldsymbol{x}\|_2 \ \ (\boldsymbol{x}, \boldsymbol{y} \in \mathbb{R}^3)$.

**Definition 3** (Distance function)**.** *$f : \mathbb{R}^3 \to [0, +\infty)$ is a distance function if*

$$f(\boldsymbol{p}) = d(\boldsymbol{p}, \{f \equiv 0\}) \quad (\forall\, \boldsymbol{p} \in \mathbb{R}^3)\ .$$

**Example.** The distance function of the unit sphere is

$$f_{\text{sphere}}(\boldsymbol{p}) = d\big(\boldsymbol{p}, \mathcal{S}_1(\boldsymbol{0})\big) = \max\left(\|\boldsymbol{p}\|_2 - 1,\, 0\right) \ (\boldsymbol{p} \in \mathbb{R}^3)\ .$$

**Definition 4** (Unbounding sphere)**.** *The unbounding sphere for the distance function $f : \mathbb{R}^3 \to [0, +\infty)$ at $\boldsymbol{p} \in \mathbb{R}^3$ is the open neighbourhood $S_{f(\boldsymbol{p})}(\boldsymbol{p})$.*

It follows from Definition 3 that there are no surface points closer to $\boldsymbol{p}$ than $f(\boldsymbol{p})$, i.e. $\mathcal{S}_{f(\boldsymbol{p})}(\boldsymbol{p}) \cap \{f \equiv 0\} = \emptyset$ .

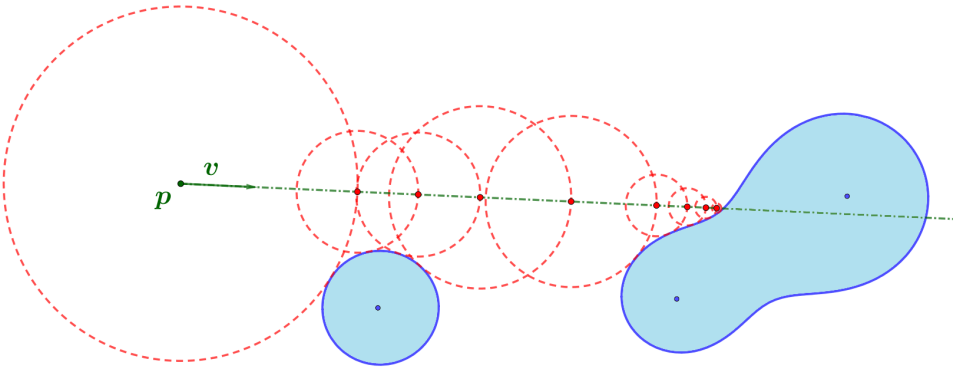



Figure 2: The sphere tracing algorithm takes distance sized steps, thereby it does not overstep a solution, yet it converges quickly. Each step defines an unbounding sphere that is disjoint from the surface.

**Algorithm 2** Sphere tracing a surface defined by a distance function

**Input:** Ray defined by $\boldsymbol{p}$ and $\boldsymbol{v} \in \mathbb{R}^3$, where $\|\boldsymbol{v}\|_2 = 1$

**Input:** Distance function $f : \mathbb{R}^3 \to \mathbb{R}$

**Output:** $t \in [0, +\infty)$ distance traveled along the ray

```
t := 0;   i := 0;
for i < i_max and f(p + t · v) > ε do
    t := t + f(p + t · v);
    i := i + 1;
end for
```

This property shows that sphere tracing shown in Algorithm 2 can be used to find the first ray-surface intersection robustly. The algorithm iteratively takes distance-sized steps along the ray; thus no ray-surface intersection is skipped while large empty spaces are traversed quickly.

As a consequence of the above, as we approach the surface along the ray, the distance to the surface cannot change more than what we have travelled. We generalize this using the Lemma 2 and Corollary 1 below.

**Lemma 2.** *Let the set $A \subseteq \mathbb{R}^n$ be a closed set and $\boldsymbol{x}, \boldsymbol{y} \in \mathbb{R}^n$. Then*

$$\big|d(\boldsymbol{x}, A) - d(\boldsymbol{y}, A)\big| \leq d(\boldsymbol{x}, \boldsymbol{y}) \ .$$

*Proof.* Since $A$ is a closed set, there exist $\boldsymbol{x}', \boldsymbol{y}' \in A$ such that $d(\boldsymbol{x}, \boldsymbol{x}') = d(\boldsymbol{x}, A)$ and $d(\boldsymbol{y}, \boldsymbol{y}') = d(\boldsymbol{y}, A)$ according to Lemma 1. Using the definition of the distance, we provide a lower bound to $d(\boldsymbol{x}, \boldsymbol{y}')$ and $d(\boldsymbol{y}, \boldsymbol{x}')$ respectively. The upper bound

is given by the triangle inequality in the $\boldsymbol{xyy'}$ and $\boldsymbol{yxx'}$ triangles, respectively:

$$d(\boldsymbol{x}, \boldsymbol{x'}) \leq d(\boldsymbol{x}, \boldsymbol{y'}) \leq d(\boldsymbol{x}, \boldsymbol{y}) + d(\boldsymbol{y}, \boldsymbol{y'}) \,, \tag{1}$$

$$d(\boldsymbol{y}, \boldsymbol{y'}) \leq d(\boldsymbol{y}, \boldsymbol{x'}) \leq d(\boldsymbol{x}, \boldsymbol{y}) + d(\boldsymbol{x}, \boldsymbol{x'}) \,. \tag{2}$$

Using (1) for the upper bound and (2) for the lower bound of $d(\boldsymbol{x}, \boldsymbol{x'})$ we have:

$$d(\boldsymbol{y}, \boldsymbol{y'}) - d(\boldsymbol{x}, \boldsymbol{y}) \leq d(\boldsymbol{x}, \boldsymbol{x'}) \leq d(\boldsymbol{y}, \boldsymbol{y'}) + d(\boldsymbol{x}, \boldsymbol{y}) \,.$$

This proves Lemma 2. □

**Definition 5** (Lipschitz constant)**.** *Let the function $f : \mathbb{R}^3 \to \mathbb{R}$ be arbitrary, we define the set of Lipschitz constants as*

$$\operatorname{Lip} f := \left\{ L > 0 \;:\; \forall \boldsymbol{x}, \boldsymbol{y} \in \mathbb{R}^3 : |f(\boldsymbol{x}) - f(\boldsymbol{y})| \leq L \cdot d(\boldsymbol{x}, \boldsymbol{y}) \right\} . \tag{3}$$

*The function $f$ is Lipschitz continuous if* $\operatorname{Lip} f \neq \emptyset$.

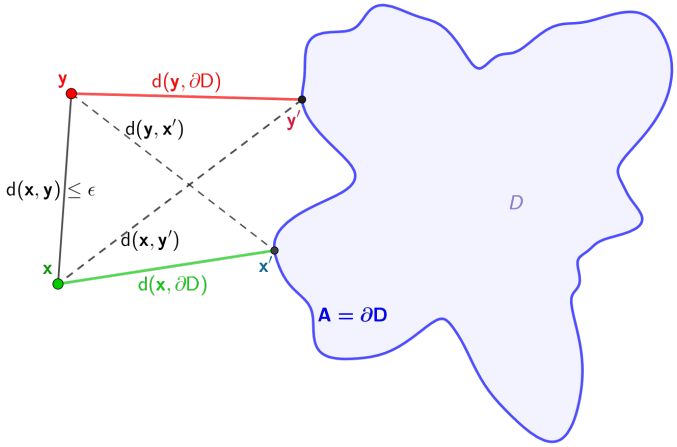



Figure 3: A visualization for the proof of Lemma 2 and Proposition 1.

**Corollary 1.** *Every signed distance function is Lipschitz continuous and their smallest Lipschitz constant is 1. Formally:*

$$\forall\, f : \mathbb{R}^3 \to \mathbb{R} \;\; \mathit{SDF} \;\; : \inf \operatorname{Lip} f = \min \operatorname{Lip} f = 1 \,.$$

*Proof.* First, the Lemma 2 above implies that $\operatorname{Lip} f \geq 1$ element-wise with $D := A$. Second $1 \in \operatorname{Lip} f$, because if $\boldsymbol{y} := \boldsymbol{x'}$, then $\boldsymbol{y} = \boldsymbol{x'} = \boldsymbol{y'} \in A$ in the proof, then inequalities turn to equities in Equation 1. □

# 5 Signed Distance Functions

**Definition 6** (SDF)**.** *If $f : \mathbb{R}^3 \to \mathbb{R}$ is continuous and $|f|$ is a distance function, then $f$ is a **s**igned **d**istance **f**unction.*

Signed distance functions (SDFs) can represent an entire volume by classifying the points of $\mathbb{R}^3$ belonging to its 'interior' ($\{f < 0\}$), 'exterior' ($\{f > 0\}$), or to the surface ($\{f \equiv 0\}$). For example, $\mathbb{R}^3 \ni \boldsymbol{p} \to \|\boldsymbol{p}\|_2 - 1 \in [-1, +\infty)$ is a signed distance function of the unit sphere.

Note that distance functions are a subset of SDFs, but they cannot differentiate between interior and surface points. For signed distance functions, we give the following equivalent definition:

**Proposition 1** (SDF equivalence)**.** *The function $f : \mathbb{R}^3 \to \mathbb{R}$ is a signed distance function if, and only if there exists a $\emptyset \neq D \subseteq \mathbb{R}^3$ set for which*

$$f(\boldsymbol{p}) = \begin{cases} d(\boldsymbol{p}, \partial D) & \text{if } \boldsymbol{p} \notin D \\ -d(\boldsymbol{p}, \partial D) & \text{if } \boldsymbol{p} \in D \end{cases} . \tag{4}$$

*Proof.* First, let us assume that $f$ is defined according to equation (4). In this case, it follows that $|f|$ is a distance function of the $\partial D = \{f \equiv 0\}$ set. Using Lemma 2 with $A := \partial D$, with $\boldsymbol{x}, \boldsymbol{y} \in \{f \geq 0\} \subseteq \mathbb{R}^3$ we know that

$$\big|f(\boldsymbol{x}) - f(\boldsymbol{y})\big| = \big|d(\boldsymbol{x}, \partial D) - d(\boldsymbol{y}, \partial D)\big| \leq d(\boldsymbol{x}, \boldsymbol{y}),$$

and therefore, $f$ is uniformly continuous function on the set $\{f \geq 0\}$. One can analogously show that $f$ is continuous on the set $\{f \leq 0\}$.

Assuming that $|f|$ is a distance function where $f : \mathbb{R}^3 \to \mathbb{R}$ is a continuous function, we have to show that the $D := \{f \leq 0\}$ set satisfy equation (4). It indeed does, because $\partial D = \{f \equiv 0\}$, and $|f(\boldsymbol{p})| = d\big(\boldsymbol{p}, \{f \equiv 0\}\big)$, and if, for example, $f(\boldsymbol{p}) > 0$, then $f(\boldsymbol{p}) = d(\boldsymbol{p}, \partial D)$ and $\boldsymbol{p} \notin D$. □

Hart [10] defined signed distance functions that are distance functions in absolute value. Definition 1 is similar to that of Hart, but the represented object $D$ appears in it. Moreover, the sign is not allowed to jump on the same side of the surface, so there is a distinct "inside" and "outside" region associated with the surface. However, this intuitive definition lacks the simplicity of the original, hence the need for Definition 6.

# 6 Offset theorem

Let us investigate the geometric operation of offsetting on SDF representations.

**Definition 7** (Offset surface)**.** *The offset surface at signed distance $c \in \mathbb{R}$ of the surface defined by the SDF $f : \mathbb{R}^3 \to \mathbb{R}$ is the $\{f \equiv c\}$ (level-)set.*

Intuitively, offsets are obtained by inflating or deflating an initial volume by some fixed radius $c \in \mathbb{R}$. Contrary to the naive assumption, however, offsets cannot be represented by $f(\boldsymbol{x}) - c = 0$ in general, see the counterexample on Figure 5. Nevertheless, there's a subset of $\mathbb{R}^3$ where the SDF of the offset can be written this way, as shown in Theorem 1.

First, we define strict convexity. Strictly convex Banach spaces include $\mathbb{R}^n$, $\mathbb{C}^n$, and $\mathrm{L}^p$ spaces with $p$-norms, if $1 < p < +\infty$.

**Definition 8** (Strictly convex normal space)**.** *The $(X, \|\cdot\|)$ normal space is strictly convex, if for all $\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{z} \in X$, the following holds:*

$$d(\boldsymbol{x}, \boldsymbol{z}) + d(\boldsymbol{z}, \boldsymbol{y}) = d(\boldsymbol{x}, \boldsymbol{y}) \iff \exists\, \lambda \in [0, 1] \ : \ \boldsymbol{z} = (1 - \lambda) \cdot \boldsymbol{x} + \lambda \cdot \boldsymbol{y} \ ,$$

*where $d(\boldsymbol{x}, \boldsymbol{y})$ denotes the induced metric, i.e. $d(\boldsymbol{x}, \boldsymbol{y}) := \|\boldsymbol{y} - \boldsymbol{x}\|$ $(\boldsymbol{x}, \boldsymbol{y} \in X)$.*

Second, the definition of the open offset set follows, which is a generalization of neighborhood in Definition 2.

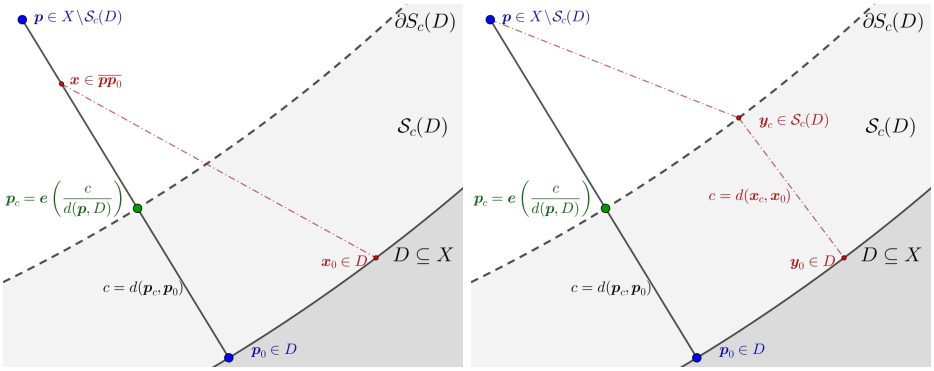
**Definition 9** (Offset set)**.** *For any $D \subseteq X$ in the metric space $(X, d)$, one can define an open offset set from $D$ with $r \geq 0$ range, as*

$$\mathcal{S}_r(D) := \{\boldsymbol{x} \in X \ : \ d(\boldsymbol{x}, D) < r\} \ .$$

Finally, we present the main contribution of this paper in the following

**Theorem 1** (Offset theorem)**.** *Let $(X, \|\cdot\|)$ be a strictly convex Banach space and $D \subseteq X$ closed. Then for any $c \geq 0$,*

$$\forall\, \boldsymbol{p} \in X \setminus \mathcal{S}_c(D) \quad : \quad d(\boldsymbol{p}, D) - c = d\big(\boldsymbol{p}, \mathcal{S}_c(D)\big) \ . \tag{5}$$




(a) Proof of $d(\boldsymbol{p}, D) - c \geq d\big(\boldsymbol{p}, \mathcal{S}_c(D)\big)$ (b) Proof of $d(\boldsymbol{p}, D) - c \leq d\big(\boldsymbol{p}, \mathcal{S}_c(D)\big)$

Figure 4: A visualization of the proof for the offset theorem.

*Proof.* Since the set containing the single element $\{\boldsymbol{p}\} \subset \mathbb{R}^3$ is compact and $D$ is closed, the extremal points exist between the two sets according to Lemma 1:

$$\exists\, \boldsymbol{p}_0 \in D \;:\; d(\boldsymbol{p}, D) = d(\boldsymbol{p}, \boldsymbol{p}_0)\;.$$

The $\boldsymbol{e}(t) := (1-t)\cdot \boldsymbol{p}_0 + t \cdot \boldsymbol{p} \in X$, $(t \in [0,1])$ is the parametric form of the $\overline{\boldsymbol{p}_0\boldsymbol{p}}$ line segment. First we show that

$$\forall\, \boldsymbol{x} \in \overline{\boldsymbol{p}_0\boldsymbol{p}} \;:\; d(\boldsymbol{x}, \boldsymbol{p}_0) = d(\boldsymbol{x}, D)\;.$$

Let us prove this by contradiction: let $\boldsymbol{x}_0 \in D$ such that $d(\boldsymbol{x}, \boldsymbol{x}_0) < d(\boldsymbol{x}, \boldsymbol{p}_0)$. Using the definition of distance to the set, the triangle inequality in $\boldsymbol{x}\boldsymbol{x}_0\boldsymbol{p}_0$, the indirect assumption, and the strict concavity, in order, we have the following:

$$\begin{aligned} d(\boldsymbol{p}, D) \;&\leq d(\boldsymbol{p}, \boldsymbol{x}_0) \;\leq\; d(\boldsymbol{p}, \boldsymbol{x}) + d(\boldsymbol{x}, \boldsymbol{x}_0) \\ &< d(\boldsymbol{p}, \boldsymbol{x}) + d(\boldsymbol{x}, \boldsymbol{p}_0) = d(\boldsymbol{p}, \boldsymbol{p}_0) = d(\boldsymbol{p}, D) \end{aligned}$$

Which is a contradiction, so all $\boldsymbol{x} \in \overline{\boldsymbol{p}_0\boldsymbol{p}}$, the $\boldsymbol{p}_0$ is a closest point in $D$. When $\boldsymbol{x} = \boldsymbol{e}(t)$, one can deduce that the distance from $D$ along $\boldsymbol{e}$ is linear:

$$d(\boldsymbol{e}(t), D) \;=\; d(\boldsymbol{e}(t), \boldsymbol{p}_0) = t \cdot d(\boldsymbol{p}, \boldsymbol{p}_0) \qquad (t \in [0,1])\;. \tag{6}$$

Because $0 \leq c \leq d(\boldsymbol{p}, D)$, $\boldsymbol{p}_c := \boldsymbol{e}\left(\dfrac{c}{d(\boldsymbol{p}, \boldsymbol{p}_0)}\right) \in \overline{\boldsymbol{p}_0\boldsymbol{p}}$. Then

$$\{\boldsymbol{p}_c\} \;=\; \partial\mathcal{S}_c(D) \;\cap\; \overline{\boldsymbol{p}_0\boldsymbol{p}}$$

because the offset surface $\partial\mathcal{S}_c(D) = \{\boldsymbol{x} \in X \;:\; d(\boldsymbol{x}, D) = c\}$ contains $\boldsymbol{p}_c$ since $d(\boldsymbol{p}_c, D) = c$; moreover, $[0,1] \ni t \to d(\boldsymbol{e}(t), D)$ function is strictly increasing, so the intersection is unique. This implies half of the proposed equality (5), because

$$d\big(\boldsymbol{p}, \mathcal{S}_c(D)\big) = d\big(\boldsymbol{p}, \partial\mathcal{S}_c(D)\big) \leq d(\boldsymbol{p}, \boldsymbol{p}_c) = d(\boldsymbol{p}, \boldsymbol{p}_0) - d(\boldsymbol{p}_0, \boldsymbol{p}_c) = d(\boldsymbol{p}, D) - c\;.$$

For the other direction, let us assume indirectly that $d(\boldsymbol{p}, \mathcal{S}_c(D)) < d(\boldsymbol{p}, \boldsymbol{p}_c)$, so there exist an $\boldsymbol{y}_c \in \mathcal{S}_c(D)$ such that $d(\boldsymbol{p}, \boldsymbol{y}_c) < d(\boldsymbol{p}, \boldsymbol{p}_c)$ as it is shown on Figure 4b. Since $D$ is a closed set, $\boldsymbol{y}_c$ also has a closest point in $D$ that we denote $\boldsymbol{y}_0 \in D$. Using the definition for the distance, the triangle inequality in $\boldsymbol{y}_c\boldsymbol{y}_0\boldsymbol{p}$, the indirect assumption, and that

$$d(\boldsymbol{y}_c, D) = d(\boldsymbol{y}_c, \boldsymbol{y}_0) \leq c\;,$$

we arrive at a contradiction:

$$\begin{aligned} d(\boldsymbol{p}, D) \;&\leq d(\boldsymbol{p}, \boldsymbol{y}_0) \;\leq\; d(\boldsymbol{p}, \boldsymbol{y}_c) + d(\boldsymbol{y}_c, \boldsymbol{y}_0) \\ &< d(\boldsymbol{p}, \boldsymbol{p}_c) + c = d(\boldsymbol{p}, \boldsymbol{p}_0) = d(\boldsymbol{p}, D)\;. \end{aligned}$$

□

**Remark.** i). Because equation (6) is generally false for $t \notin [0,1]$, $\boldsymbol{p}$ must not be inside $\mathcal{S}_c(D)$.

ii). Note that the proof does not require that the closest point $\boldsymbol{p}_0$ to be unique, any one of them will suffice.

iii). Consider the signed distance function form of this theorem, Corollary 2. Because of equation (6), if $f$ is differentiable at point $\boldsymbol{x} \in \overline{\boldsymbol{p}\boldsymbol{p}_0} \subseteq \mathbb{R}^3$, then $\nabla f(\boldsymbol{x}) = \frac{\boldsymbol{p}-\boldsymbol{p}_0}{\|\boldsymbol{p}-\boldsymbol{p}_0\|_2}$.

We can now state the theorem on offsetting SDFs:

**Corollary 2** (Offset of an SDF)**.** *If $f : \mathbb{R}^3 \to \mathbb{R}$ is an SDF, then for any $0 \neq c \in \mathbb{R}$ offset, the function $f - c$ is an SDF on the set $\left\{\frac{f}{c} \geq 1\right\}$.*

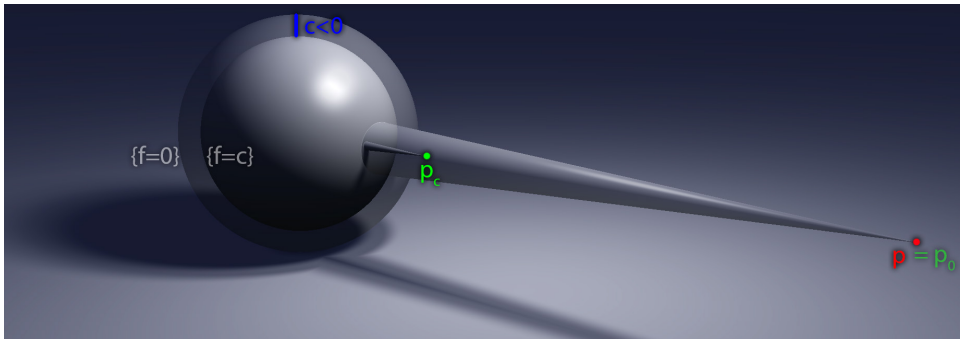



Figure 5: A counterexample for Corollary 2, when the condition does not hold.

**Remark.** i). $\left\{\frac{f}{c} \geq 1\right\} = \begin{cases} \{f \leq c\} & \text{if } c < 0 \\ \{f \geq c\} & \text{if } c > 0 \end{cases}$.

ii). The theorem is untrue for other points, as a counterexample is demonstrates this on Figure 5. Let $c < 0$, and $\boldsymbol{p}$ be a point on a highly convex point on the surface as seen on the figure, so $\{f \equiv 0\} \ni \boldsymbol{p} \notin \left\{\frac{f}{c} \geq 0\right\}$. Then, let $\boldsymbol{p}_0$ be a closest point to $\boldsymbol{p}$ on the original surface $\{f \equiv 0\}$, and $\boldsymbol{p}_c$ be the closest point on offset surface $\{f \equiv c\}$. Clearly $\boldsymbol{p} = \boldsymbol{p}_0$, but because of the said convexity, $|c| < d(\boldsymbol{p}_0, \boldsymbol{p}_c) = d(\boldsymbol{p}, \{f \equiv c\})$; and therefore, $d(\boldsymbol{p}, \{f \equiv 0\}) - c = -c \neq d(\boldsymbol{p}, \{f \equiv c\})$.

# 7 Conclusion

This paper presented a theoretical overview of surfaces defined by signed distance functions. We formulated equivalent definitions to emphasize the geometric properties of this implicit representation.

We defined an abstract offset set of an arbitrary set in Banach spaces. Our main theoretical contribution is a theorem stating a distance equivalence for points outside of the offset set.

Most importantly, Theorem 1 exposes a way to compute a signed distance function of an offset surface defined by an SDF by merely subtracting the offset radius from the function. However, this formulation is limited to the exterior of the offset volume, and the error can be arbitrarily large as we demonstrated on Figure 5.

The simple subtraction formula for offsetting a signed distance function was often used in practice, but it was only validated empirically. Our paper gave this missing guarantee and explained when this formula does not work.

# References

[1] Aaltonen, Sebastian. GPU-based clay simulation and ray-tracing tech in Claybook. San Francisco, CA, March 2018. Game Developers Conference.

[2] Angles, Baptiste, Tarini, Marco, Wyvill, Brian, Barthe, Loïc, and Tagliasacchi, Andrea. Sketch-based implicit blending. *ACM Trans. Graph.*, 36(6):181:1–181:13, November 2017. DOI: 10.1145/3130800.3130825.

[3] Bernhardt, Adrien, Barthe, Loic, Cani, Marie-Paule, and Wyvill, Brian. Implicit blending revisited. In *Computer Graphics Forum*, volume 29, pages 367–375. Wiley Online Library, 2010.

[4] Bloomenthal, Jules and Wyvill, Brian, editors. *Introduction to Implicit Surfaces*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1997.

[5] Dyer, R.H. and Edmunds, D.E. *From Real to Complex Analysis*. Springer Undergraduate Mathematics Series. Springer International Publishing, 2014.

[6] Evans, Alex. Learning from failure: a survey of promising, unconventional and mostly abandoned renderers for 'dreams ps4', a geometrically dense, painterly ugc game. In *Advances in Real-Time Rendering in Games*. MediaMolecule, SIGGRAPH, 2015.

[7] Gomes, Abel, Voiculescu, Irina, Jorge, Joaquim, Wyvill, Brian, and Galbraith, Callum. *Implicit Curves and Surfaces: Mathematics, Data Structures and Algorithms*. Springer Publishing Company, Incorporated, 1st edition, 2009.

[8] Gourmel, Olivier, Pajot, Anthony, Paulin, Mathias, Barthe, Loic, and Poulin, Pierre. Fitted BVH for Fast Raytracing of Metaballs. *Computer Graphics Forum*, 2010. DOI: 10.1111/j.1467-8659.2009.01597.x.

[9] Hansen, C., Hijazi, Y., Hagen, H., Knoll, A., and Wald, I. Interactive ray tracing of arbitrary implicits with simd interval arithmetic. In *IEEE/ EG Symposium on Interactive Ray Tracing 2007(RT)*, volume 00, pages 11–18, 09 2007. DOI: 10.1109/RT.2007.4342585.

[10] Hart, John C. Sphere tracing: A geometric method for the antialiased ray tracing of implicit surfaces. *The Visual Computer*, 12:527–545, 1994.

[11] Hutchinson, John E. *Introduction to Mathematical Analysis*. Australian National University Lecture Notes, 1994.

[12] Malladi, Ravikanth, Sethian, James A., and Vemuri, Baba C. Shape modeling with front propagation: A level set approach. *IEEE Trans. Pattern Anal. Mach. Intell.*, 17(2):158–175, February 1995. DOI: 10.1109/34.368173.

[13] Pasko, A., Adzhiev, V., Sourin, A., and Savchenko, V. Function representation in geometric modeling: concepts, implementation and applications. *The Visual Computer*, 11(8):429–446, Aug 1995. DOI: 10.1007/BF02464333.

[14] Ricci, A. A Constructive Geometry for Computer Graphics. *The Computer Journal*, 16(2):157–160, May 1973. DOI: http://dx.doi.org/10.1093/comjnl/16.2.157.

[15] Shapiro, Vadim. Semi-analytic geometry with r-functions. *Acta Numerica*, 16:239–303, 2007. DOI: 10.1017/S096249290631001X.

[16] Sherstyuk, Andrei. Fast ray tracing of implicit surfaces. *Computer Graphics Forum*, 18(2):139–147. DOI: 10.1111/1467-8659.00364.

[17] Szécsi, László and Illés, Dávid. Real-time metaball ray casting with fragment lists. In Andújar, Carlos and Puppo, Enrico, editors, *Eurographics (Short Papers)*, pages 93–96. Eurographics Association, 2012.

[18] Wright, Daniel. Dynamic occlusion with signed distance fields. In *Advances in Real-Time Rendering in Games*. Epic Games (Unreal Engine), SIGGRAPH, 2015.

[19] Wyvill, Brian, Guy, Andrew, and Galin, Eric. Extending the CSG Tree. Warping, Blending and Boolean Operations in an Implicit Surface Modeling System. *Computer Graphics Forum*, 1999. DOI: 10.1111/1467-8659.00365.

Acta Cybernetica **24** (2019) 29–41.

# Multi Party Computation Motivated by the Birthday Problem*

Péter Hudoba[a] and Péter Burcsi[b]

**Abstract**

Suppose there are $n$ people in a classroom and we want to decide if there are two of them who were born on the same day of the year. The well-known birthday paradox is concerned with the probability of this event and is discussed in many textbooks on probability. In this paper we focus on cryptographic aspects of the problem: how can we decide if there is a collision of birthdays without the participants disclosing their respective date of birth. We propose several procedures for solving this generally in a privacy-preserving way and compare them according to their computational and communication complexity.

**Keywords:** secure multi-party computation, birthday paradox, privacy-preserving, communication complexity

## 1 Introduction

### 1.1 Description of the problem

The birthday paradox or birthday problem [14, 18, 1] investigates the following question: $n$ people are selected at random from a large population. What is the probability that at least $r$ people share the same birthday? It's usually referred to as a paradox because of the unintuitively large probability of over 50% already for the relatively small value of $n = 23$ and $r = 2$.

In the present paper we focus on cryptographic aspects of the problem. We examine whether and how the $n$ participants can decide if $r$ of them share the same date of birth without any of them publicly announcing his or her birthday,

*Péter Hudoba's was work supported by EFOP-3.6.3-VEKOP-16-2017-00001: Talent Management in Autonomous Vehicle Control Technologies. The Project is supported by the Hungarian Government and co-financed by the European Social Fund. Péter Burcsi's work was supported by the European Union, co-financed by the European Social Fund (EFOP-3.6.2-16-2017-00013, Thematic Fundamental Research Collaborations Grounding Innovation in Informatics and Infocommunications).

[a]Eötvös Loránd University, Budapest, Hungary, E-mail: peter.hudoba@inf.elte.hu

[b]ELTE 3in External Research Group, E-mail: bupe@inf.elte.hu

DOI: 10.14232/actacyb.24.1.2019.4

using secure communication. This is a so-called multi-party computation, see e.g. Chapter 7 of [9] or [10].

The $n = 2$ case is well-known and named Tiercé or socialist millionaires' problem. This is similar to Yao's millionaires' problem originally introduced in [19, 20] where the two participants want to compare their secrets (decide which one is larger). Later, other solutions were proposed, e.g. [5],[13],[16], [15] but all of them consider the case of 2 participants.

A first idea would be to use pairwise socialist millionaires' protocols for the general $n$-participant version. However, in case of equality the two participants involved would instantly learn each other's secrets which we want to avoid when $n \geq 3$. In what follows, we deal with the $r = 2$ case but general $n$.

More generally and formally we have a finite but possibly large set of possible values $V$ (corresponding to possible birthdays) and each of $n$ participants holding a secret value $x_i \in V$ (their respective birthdays). We want to compute, using a secure multi-party computation, the following function:

$$f(x_1, \ldots, x_n) = \begin{cases} 1, & \text{if } \exists i, j \in \{1, \ldots, n\} : i \neq j \wedge x_i = x_j \\ 0, & \text{otherwise} \end{cases} \tag{1}$$

## 1.2 Security assumptions and comparison of protocols

In this paper we make the assumption on the participants' behavior called *honest but courious* or semi-honest. The participants are honest in following the protocol which means they do not poison or dilate the data, but if they can gain information without poisoning the algorithm, they will do it. With these conditions we want to make sure no one learns any other participant's secret.

We can characterize the level of privacy of a secure multi-party scheme with numbers $adv_a$ (respectively $adv_p$) corresponding to the minimal number of active (resp. passive) coordinated adversary participants who are able to gain access to secrets of the others, while still following the protocol. Below, when we call a scheme "$adv$ out of $n$" scheme, we'll always mean $adv_a = adv$.

At the end of each section, we briefly discuss the running time and communication complexity of the scheme. We always consider only the slowest participant, unless the others are idle. We also restrict our attention to data sending (rather than receiving), because all communication is symmetric in all of the described algorithms.

We will express the running time in terms of basic operations, using the following notations. $T(M, l)$ and $T(A, l)$ are the running time in order of the multiplication and the adding for $l$-word unsigned integers. $T(C, l)$ is the running time of sending a $l$-word message, $T(R, l)$ is the running time of generating a $l$-word random number. As a shorthand for integers that fit in one word we write: $T(M) = T(M, 1)$, $T(C) = T(C, 1)$, $T(R) = T(R, 1)$. Finally, $W$ is the number of bits in one word.

# 2 Multi-party protocols for the birthday problem

## 2.1 Voting based

Below, by voting protocol we mean a multi-party computation where each participant casts a 'yes' or 'no' vote, and the protocol computes the number of 'yes' and 'no' votes. The birthday problem can be solved using voting protocols [6]. In a naïve approach, we perform a voting for all possible values in $V$. Whenever a value receives more than one vote, we know there is a collision. Unfortunately, this is unfeasible when $|V|$ is large (for birthdays it could still work).

In order to improve the efficiency of the approach, we can partition the set $V$ of possible values into subsets $S_i$, which we call *slots*. First we perform the votes for the subsets and then focus on values from those subsets $S_i$ that have received at least two votes. This approach can reduce the number of the required voting rounds. Clearly, if the number of slots is too small, then there might be a lot of slots with at least two values and we have to test all values in these slots. On the other hand, if the slots are two small, then the number of slots is not much smaller than the number of possible values.

In the following we analyze the possible slot numbers in worst and average cases. We denote the number of possible values by $k \in \mathbb{Z}^+$, the number of participants by $n \in \mathbb{Z}^+$ and the number of slots by $q \in \mathbb{Z}^+$. We try to distribute the possible values among the slots as equally as possible and analyze the optimal choice of parameter $q$.

**WorstCase** If we distribute the participant values equally to the slots, each slot will contain $\left\lceil \frac{k}{q} \right\rceil$ or $\left\lfloor \frac{k}{q} \right\rfloor$ values. Let's call the slots that have $\left\lceil \frac{k}{q} \right\rceil$ values "full" slots. Denote the number of full slots by $T$. Then

$$T = \begin{cases} q, & \text{if } q \mid k \\ k - q\left\lfloor \frac{k}{q} \right\rfloor, & \text{otherwise} \end{cases}$$

The maximal number of slots with at least 2 participant values is $r = min\{\left\lfloor \frac{n}{2} \right\rfloor, q\}$. Denote the maximal number of full slots with at least 2 participant values by $T_r = min\{r, T\}$. The number of necessary voting rounds in the worst case is $q + T_r \left\lceil \frac{k}{q} \right\rceil + (r - T_r) \cdot \left\lfloor \frac{k}{q} \right\rfloor$.

In the case when $q \mid k$, we have $T = q$, so $T_r = min\{r, q\} = min\{min\{\left\lfloor \frac{n}{2} \right\rfloor, q\}, q\} = min\{\left\lfloor \frac{n}{2} \right\rfloor, q\}$. We get $r - T_r = 0$, so the number of voting rounds is $q + min\{\left\lfloor \frac{n}{2} \right\rfloor, q\}\frac{k}{q} = q + k \cdot min\{\left\lfloor \frac{n}{2} \right\rfloor \frac{1}{q}, 1\}$. The derivative w.r.t. $q$ is $1 + k \cdot min\{\left\lfloor \frac{n}{2} \right\rfloor (-\frac{1}{q^2}), 0\}$ showing that in case of $\left\lfloor \frac{n}{2} \right\rfloor \geq q$ we do not have an optimal value of $q$. If $\left\lfloor \frac{n}{2} \right\rfloor \leq q$, then we have a minimum at $q = \sqrt{k \cdot \left\lfloor \frac{n}{2} \right\rfloor}$.

If we assume that $q \nmid k$, then $T = k - q\left\lfloor \frac{k}{q} \right\rfloor$, so $T_r = min\{r, T\} = min\{r, k - q\left\lfloor \frac{k}{q} \right\rfloor\} = min\{min\{\left\lfloor \frac{n}{2} \right\rfloor, q\}, k - q\left\lfloor \frac{k}{q} \right\rfloor\} = min\{\left\lfloor \frac{n}{2} \right\rfloor, k - q\left\lfloor \frac{k}{q} \right\rfloor\}$. In this case the formula for the rounds gives $q + T_r\left\lceil \frac{k}{q} \right\rceil + (r - T_r) \cdot \left\lfloor \frac{k}{q} \right\rfloor = q + T_r(\left\lceil \frac{k}{q} \right\rceil - \left\lfloor \frac{k}{q} \right\rfloor) + r \cdot \left\lfloor \frac{k}{q} \right\rfloor = q + T_r + r \cdot \left\lfloor \frac{k}{q} \right\rfloor = q + min\{\left\lfloor \frac{n}{2} \right\rfloor, k - q\left\lfloor \frac{k}{q} \right\rfloor\} + min\{\left\lfloor \frac{n}{2} \right\rfloor, q\} \cdot \left\lfloor \frac{k}{q} \right\rfloor$. Below we approximate $q + \left\lfloor \frac{k}{q} \right\rfloor$ by $q + \frac{k}{q}$ in order to simplify the calculation.

- If $\left\lfloor \frac{n}{2} \right\rfloor \geq q$, then $q + min\{\left\lfloor \frac{n}{2} \right\rfloor, k - q\left\lfloor \frac{k}{q} \right\rfloor\} + q \cdot \left\lfloor \frac{k}{q} \right\rfloor$. If we remove the floor functions then the derivative is 1 so the minimum is at one of the boundaries.
- If $\left\lfloor \frac{n}{2} \right\rfloor \leq q$, then $q + min\{\left\lfloor \frac{n}{2} \right\rfloor, k - q\left\lfloor \frac{k}{q} \right\rfloor\} + \left\lfloor \frac{n}{2} \right\rfloor \cdot \left\lfloor \frac{k}{q} \right\rfloor$. The derivative after removing floor functions is $1 + min\{0, 0\} - \left\lfloor \frac{n}{2} \right\rfloor \cdot \frac{k}{q^2}$, so the minimum is at $q = \sqrt{k \cdot \left\lfloor \frac{n}{2} \right\rfloor}$ which is usually better than the first case.

**Average case** We compute the expected number of slots with at least two participant values. This can be formulated as follows. Let $f : A \to B$ where $|A| = n$, $|B| = q$, $q < n$, $f$ chosen uniformly among all such functions. We are interested in $E(\#\{b \in B | \left|f^{-1}(b)\right| > 1\}) = \sum_{b \in B} P(\left|f^{-1}(b)\right| > 1) = |B| \cdot P\left(\left|f^{-1}(b_1)\right|\right)$, where $b_1$ denotes the first slot. $P\left(\left|f^{-1}(b_1)\right|\right) = 1 - P(\left|f^{-1}(b_0)\right| = 1) - P(\left|f^{-1}(b_0)\right| = 0) = 1 - n\left(\frac{q-1}{q}\right)^{n-1}\frac{1}{q} - \left(\frac{q-1}{q}\right)^n$. We approximate this by $1 - \frac{n}{q}\left(\frac{1}{e}\right)^{\frac{n}{q}} - \left(\frac{1}{e}\right)^{\frac{n}{q}} = 1 - \frac{n+q}{q}\left(\frac{1}{e}\right)^{\frac{n}{q}}$. So the estimated expected number of slots with at least 2 values is: $q\left(1 - \frac{n+q}{q}e^{-\frac{n}{q}}\right) = q - (n+q)e^{-\frac{n}{q}}$. The number of voting rounds is $q + k/q\left(q - (n+q)e^{-\frac{n}{q}}\right)$. Deriving and solving for zero we get $n^2/q^3 = ke^{n/q}$ and thus we can compute the optimal choice for $q$.

If we assume that $\left\lfloor \frac{n}{2} \right\rfloor \leq q$ and $q \mid k$, then the worst case needs $q + \left\lfloor \frac{n}{2} \right\rfloor \frac{k}{q}$ voting rounds. Since we assumed semi-honest behavior, we can use a simple voting algorithm with leader (we show runtime in parenthesis): every participant sends a fragment to two others ($2T(C)$), everyone receives two shares and adds to their remaining share ($2T(A)$) and sends the fragment of the solution to the leader node ($T(C)$) who combines all received values ($nT(A)$). The running time is $\left(q + \left\lfloor \frac{n}{2} \right\rfloor \frac{k}{q}\right)(3T(C) + 2T(A) + nT(A))$.

**Remark 1.** We can use a multiple hashing (several orthogonal sets of slots) too, if $k$ is small relative to $n$.

**Remark 2.** If user behavior is more complicated and we insist on more privacy, there are several voting protocols to be considered, e.g. [3, 6].

## 2.2 Pots

A folklore method for privately computing the average age of participants is the following. Start with one of the participants, called the seeder, putting a piece of paper containing a secret random value, the seed, into a pot. The seed could be chosen e.g. uniformly among the first one thousand positive integers. Then the participants secretly increment the value by their respective ages, one-by-one. At the end the seeder subtracts the seed and we get the sum of the ages.

We adapt this method for the birthday problem: let there be $n$ participants, $m$ seeders ($m \leq n$), and $k$ pots, initially containing 0. We start by every seeder getting the pots and adding some random number to the number found in it (independently for every pot). They remember that number for later. To each participant, we assign a pot that will be responsible for taking into account the participant's value (birthday). When inserting their seed into the pots, the seeders also increment by 1 the value in the pot holding their secret.

Next, all non-seeder participants take the pots and add 1 to the pot assigned to their secret, and 0 to the other pots. Finally, the seeders subtract the random numbers they added at the beginning. The order in which the seeders perform the final phase is shuffled compared to the initial phase in order to have different predecessors and successors for extra privacy. We can always achieve this when $n \geq 5$ (we can find two disjoint Hamiltonian cycles in the complete graph with at least 5 vertices).

The adding/subtracting functions can come from an arbitrary Abelian group, e.g. exclusive or operation on a fixed length word, or a simple unsigned integer addition/subtraction in a $\mathbb{Z}_m$. In order to detect collisions for values from a set of size $k$, we could use a bit vector of length $k$. Adding a secret value of $m$ to the pot means flipping the $m$th bit of the bit vector. If all values of the participants are distinct, then the number of the 1 bits in the final result is exactly $n$, otherwise we have flipped at least one bit back to 0, reducing the number of 1 bits.

To illustrate this method with an example, imagine that 3 people want to know if any two of them share the same favourite Star Wars movie from the original trilogy. To indicate which movie they prefer, everyone sets a bit vector of length three: 100 corresponds to the first movie, 010 to the second and 001 to the third one. The bitwise XOR of the three vectors reveals whether there is a collision: a necessary and sufficient condition for this is that the number of 1 bits is smaller than the number of participants (colliding 1s puts out each other). In order to do this with privacy preserved, everyone adds a random mask to the vectors which are then subtracted at the end.

The security level of the scheme depends on the number of seeders. If not all participants are seeders, all of the non-seeders' values can be claimed by the two neighboring participants, since they can simply calculate the difference. So this scheme is 2 out of $n$ if $m < n$. If all of the participants are seeders, but we do not use the shuffling, we get 2 out of $n$ again: in this case the neighbors can calculate the difference of the differences and get the secret. If we do use shuffling, we get a 4 out of $n$ scheme. Below, when the $m = n$ case is considered, we always mean the

shuffled version, and the non-shuffled version if $m < n$.

For the runtime analysis, observe that in the seeder phase we have to generate one random number, perform two additions (add 1 or 0 to the random number and add to the pot) and send the pot to the next seeder. This is done in $max\{k, m\}$ rounds, so the time needed is: $max\{k, m\}(T(C)+2T(A)+T(R))$. The next phase is the value filling for non-seeders: $max\{k, (n-m)\}(T(C)+T(A))$. Finally removing seeds: $max\{k, m\}(T(C) + T(A))$. The overall complexity is: $max\{k, m\}(T(C) + 2T(A) + T(R)) + max\{k, (n-m)\}(T(C) + T(A)) + max\{k, m\}(T(C) + T(A)) = max\{k, m\}(2T(C) + 3T(A) + T(R)) + max\{k, (n-m)\}(T(C) + T(A))$.

## 2.3 Big Pot

We consider the special case where we only have one pot (unsigned integer), with $k$ bits, initiated by the seeders (with random numbers). After seeding, every participant flips one bit of the pot corresponding to his or her secret. Finally the seeders remove their random numbers. If the number of one bits is not equal to the number of participants, we found a collision. In order to avoid the attack by the neighbors, it is also necessary to use $n$ seeders.

The complexity of the algorithm is the following: $m(T(C, \lceil \frac{k}{W} \rceil)+2T(A, \lceil \frac{k}{W} \rceil)+ T(R, \lceil \frac{k}{W} \rceil)) + (n-m)(T(C, \lceil \frac{k}{W} \rceil) + T(A, \lceil \frac{k}{W} \rceil)) + m(T(C, \lceil \frac{k}{W} \rceil) + T(A, \lceil \frac{k}{W} \rceil)) = (m+n)T(C, \lceil \frac{k}{W} \rceil) + (2m+n)T(A, \lceil \frac{k}{W} \rceil) + mT(R)$.

## 2.4 Additive secret sharing based

In this section we consider schemes that are based on additive secret sharing. W.l.o.g, we assume secret values are from a finite field. The secret pieces of information are split into multiple fragments and shared in the following way: every participant holding secret $x_i$ chooses 2 random numbers $x_{i,1}, x_{i,2} \in \mathbb{F}_{p^q}$ ($\mathbb{F}_{p^q}$ is a finite field with $p^q$ element, where $p$ is a prime, using the ordinary + operator) and then calculates $x_{i,3} = x_i - x_{i,1} - x_{i,2}$. Clearly $x_i = x_{i,1} + x_{i,2} + x_{i,3}$.

The problem statement (1) can be reformulated into an algebraic form (2) to better fit secret sharing.

$$f = sgn\left(\left|\prod_{i=1}^{n}\prod_{j=i+1}^{n}(x_i - x_j)\right|\right) \tag{2}$$

Clearly, the product vanishes if and only if there is a collision of values.

In the following assume that there are $n$ participants and denote the $i$th participant's secret by $x_i = x_{i,1} + x_{i,2} + x_{i,3}$, $i = 1, \ldots, n$. Two of the three shares can be distributed, because without the third share it does not give any information for an adversary. In our approach, if $q$ participants perform part of the protocol, we allow the $i$th participant to have access to shares $\{x_{j,k} \mid \forall j \in \{1..q\}, \forall k \in \{1,2,3\} : i \not\equiv k \ mod\ 3\}$.

Expanding the product in (2) gives an exponentially growing formula w.r.t. $n$, so we will relax privacy conditions and perform multiple collision-detection protocols

for smaller subsets of participants. We will consider the general collision detection protocol where collisions to be detected are given by a graph. For example, with people seated in a circle, we might only be interested in two *neighbors* having the same birthday, which corresponds to the collision-detection graph being a cycle.

If we cover all edges of the $n$-vertex complete graph by smaller collision-detection graphs (possibly redundantly), then we can detect all collisions, using several iterations on a more friendly version of (2).

We consider only simple finite and undirected graphs and will use standard graph-theoretical concepts (see e.g. [4] for graph concepts used). As usual, $K_t$ denotes a complete graph with $t$ vertices, $K_{t,u}$ denotes the complete bipartite graphs with $t$ and $u$ sized parts, $P_t$ denotes a vertex disjoint path of length $t-1$, and $S_t$ denotes the "star" graph with $t$ edges ($K_{1,t-1}$). Below we focus on how the generalized version of the socialist millionaires' protocol can be performed on small collision-detection graphs.

### 2.4.1 $SMP(K_3)$

In the 3-participant case we want to find $sgn\left(|(x_1-x_2)(x_1-x_2)(x_2-x_3)|\right)$. In Table 1 we show which shares are made available to which participant in an encrypted way (one-to-one communication).

Table 1: Shares that one participant holds

| 1. participant | | |
|---|---|---|
| $x_{1,1}$ | $x_{1,2}$ | $x_{1,3}$ |
| | $x_{2,2}$ | $x_{2,3}$ |
| | $x_{3,2}$ | $x_{3,3}$ |

| 2. participant | | |
|---|---|---|
| $x_{1,1}$ | | $x_{1,3}$ |
| $x_{2,1}$ | $x_{2,2}$ | $x_{2,3}$ |
| $x_{3,1}$ | | $x_{3,3}$ |

| 3. participant | | |
|---|---|---|
| $x_{1,1}$ | $x_{1,2}$ | |
| $x_{2,1}$ | $x_{2,2}$ | |
| $x_{3,1}$ | $x_{3,2}$ | $x_{3,3}$ |

Expanding the product, one finds that most terms can be computed by at least one participant individually. The part of the formula missing is (3).

$$2x_{12}x_{13}x_{21}-2x_{12}x_{13}x_{31}-2x_{12}x_{21}x_{23}+2x_{13}x_{31}x_{32}+2x_{21}x_{23}x_{32}-2x_{23}x_{31}x_{32} \quad (3)$$

With the help of a 4th participant, we can compute each of the summands because the necessary fragments can be sent to the helper without revealing any of the secrets. The fourth participant does not share a secret in this part.

Covering $K_n$ by copies of $K_3$ graphs is not entirely trivial. The number of copies of $K_3$ needed is trivially between $\binom{n}{2}$ and $\binom{n}{2}/3$. The latter value is obtained by disjoint copies in the case of some special values of $n$ using finite geometries. The overlapping decomposition a graph into the minimum number of complete subgraphs is NP-complete in general [11, 7]. There are polynomial time algorithms that creates cover by trees, $K_{1,k}$ or $P_4$ with overlap 2 [2]. In [17] it is proved that optimal covering is polynomial with $S_k$ and $P_k$ graphs. Covering a graph with

complete bipartite subgraphs, but not with a fixed size is discussed in [12]. The hardness of lane covering is discussed in [8]. Note that non-disjoint covers by small collision-detection graphs can leak information: if e.g. two participants detect a collision in two distinct 3-tuples with both of them involved in the collisions, the a posteriori probability of the two of them colliding increases largely.

Overall the collision detection protocol with $SMP(K_3)$ gives us an extra level of privacy compared to the pairwise socialist millionaires' protocol without adding to much computational overhead.

### 2.4.2 $SMP(P_3)$

Another approach computes only $(x_1 - x_2)(x_2 - x_3)$ for three participants, meaning we cover our complete graph with $P_3$ graphs. There is no need for a helper participant to do this type of sub protocol. The formulas for the participants can be seen in (4).

$$\begin{aligned} f1 =& x_{11}x_{23} - x_{11}x_{32} + x_{12}x_{23} - x_{12}x_{33} + \\ & x_{13}x_{22} + x_{13}x_{23} - x_{22}^2 - x_{13}x_{32} + x_{22}x_{32} + x_{23}x_{32} + x_{23}x_{33} \\ f2 =& x_{11}x_{22} - x_{11}x_{31} + x_{13}x_{21} - x_{13}x_{31} - \\ & x_{13}x_{33} - 2x_{21}x_{23} - x_{23}^2 + x_{21}x_{31} - 2x_{22}x_{23} + x_{22}x_{31} + x_{23}x_{31} \\ f3 =& x_{11}x_{21} - x_{11}x_{33} + x_{12}x_{21} + x_{12}x_{22} - \\ & x_{12}x_{31} - x_{12}x_{32} - x_{21}^2 - 2x_{21}x_{22} + x_{21}x_{32} + x_{21}x_{33} + x_{22}x_{33} \end{aligned} \tag{4}$$

**Theorem 1** (Theorem B. from [17])**.** *Let $p$ and $q$ nonnegative integers, let $n$ and $k$ be positive integers such that $n \geq 4k$ and $k(p+q) = \binom{n}{2}$, and let one of the following conditions hold:*
*(1) $k$ is even and $p \geq \frac{k}{2}$,*
*(2) $k$ is odd and $p \geq k$.*
*Then there exists a decomposition of $K_n$ into $p$ copies of $P_{k+1}$ and $q$ copies of $S_{k+1}$.*

By Theorem 1, we can prove that we can decompose a complete subgraph with at least 4 vertices into $P_3$ graphs if $4 \mid n$ or $4 \mid (n-1)$. The theorem gives the number of covering graphs $p = \frac{n(n-1)}{4}$.

If $4 \mid n$, then every participant in one round generates two random number ($2T(R)$) subtracts two to achieve the secret fragmenting ($2T(A)$), sends two fragments ($4T(C)$) to the other participants (2-2 share to each), has 11 multiplications ($11T(M)$) and additions ($11T(A)$) and finally they share the $f_i$ part of the solution to a leader in the group ($T(C)$). We have $\frac{n}{2}$ rounds, so we get $\frac{n}{2}\left(11T(M) + 13T(A) + 5T(C) + 2T(R)\right)$ for the overall running time.

## 2.5 $SMP(2K_2)$

A 4-participant approach that performs subprotocol based on $2K_2$ graphs (see Figure 1 (c)) can also solve the problem without a helper. The fourth participant's fragments can be seen in Table 2.

Table 2: Shares that one participant holds in 4 participant case

| 4. participant | | |
|---|---|---|
| | $x_{1,2}$ | $x_{1,3}$ |
| | $x_{2,2}$ | $x_{2,3}$ |
| | $x_{3,2}$ | $x_{3,3}$ |
| $x_{4,1}$ | $x_{4,2}$ | $x_{4,3}$ |

If we substitute the fragments and expand the $(x_1 - x_2)(x_3 - x_4)$ we get the participants formulas (5). Trivially a disjoint cover can be built up of the complete graph by two lines if $4 \mid n$ or $4 \mid n-1$. If $4 \mid n$, then in each round, no participant is idle.

$$\begin{aligned} f_1 =& x_{11}x_{32} - x_{11}x_{42} - x_{11}x_{43} + x_{12}x_{32} - x_{12}x_{42} \\ & - x_{22}x_{33} + x_{22}x_{42} - x_{23}x_{32} - x_{23}x_{33} \\ f_2 =& x_{13}x_{31} + x_{13}x_{33} - x_{21}x_{31} - x_{21}x_{33} + x_{21}x_{43} \\ & + x_{22}x_{41} + x_{22}x_{43} - x_{23}x_{31} + x_{23}x_{43} \\ f_3 =& x_{11}x_{31} + x_{11}x_{33} - x_{11}x_{41} + x_{12}x_{31} - x_{12}x_{41} \\ & - x_{21}x_{32} + x_{21}x_{41} + x_{21}x_{42} - x_{22}x_{31} \\ f_4 =& x_{12}x_{33} - x_{12}x_{43} + x_{13}x_{32} - x_{13}x_{41} - x_{13}x_{42} \\ & - x_{13}x_{43} + x_{23}x_{41} + x_{23}x_{42} - x_{22}x_{32} \end{aligned} \tag{5}$$

The complexity is as follows: every participant in one round generates the shares $(2T(R) + 2T(A))$, sends two fragments $(6T(C))$ for all of the other participants, and does 9 multiplications $(9T(M))$ and additions $(9T(A))$ and finally shares the $f_i$ part of the solution with a leader in the group $(T(C))$. We have $\frac{n}{2}$ rounds, so we get $\frac{n}{2}(9T(M) + 11T(A) + 7T(C) + 2T(R))$.

### 2.6 Other collision-detection graphs

We also experimented with other collision-detection graphs. We expanded the formulas for different graphs and distributed the fragments by a randomized greedy algorithm. Figure 1 shows how many participants are needed for the different graphs used.

## 3 Comparison and conclusion

Some algorithms have some restrictions on the number of participants for which they can be applied. Leakage means some information that is unavoidably leaked in case of collisions. In Table 3, we compare the algorithms by the level of privacy,

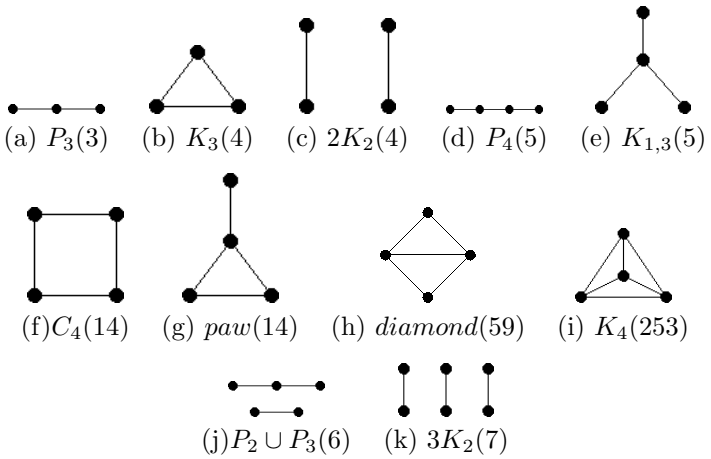



Figure 1: Graphs with number of necessary participants
**Source:** http://www.graphclasses.org/smallgraphs.html (reach: 2018-09-05)

the restrictions and show how many active adversaries in the system can claim any information of any other participant in the worst case. In Table 4 the runtimes can be seen.

Table 3: Adversary tolerance, most important information leakage and restrictions of the algorithms

| Method name | Adversary | Leaked information | Restriction |
|---|---|---|---|
| Voting based | 2 | What is the duplicated value | |
| Pots ($m = 1$) | 2 | What is the duplicated value | |
| Pots ($m = n$) | 4 | What is the duplicated value | $n \geq 5$ |
| Big pot ($m = 1$) | 2 | How many collisions exist | |
| Big pot ($m = n$) | 4 | How many collisions exist | $n \geq 5$ |
| $SMP(P_3)$ | 2 | Equality guess with $\frac{1}{2}$ probability | $4 \mid n \wedge n \geq 8$ |
| $SMP(2K_2)$ | 2 | Equality guess with $\frac{1}{2}$ probability | $4 \mid n \vee 4 \mid n-1$ |

Table 4: Estimated runtime of algorithms based on base functions (addition, multiplication, random number generation and communication)

| Method name | Runtime |
|---|---|
| Voting based | $\left(q + \left\lfloor \frac{n}{2} \right\rfloor \frac{k}{q}\right)(3T(C) + 2T(A) + nT(A))$ |
| Pots ($m = 1$) | $k(2T(C) + 3T(A) + T(R))+$ $max\{k, (n-1)\}(T(C) + T(A))$ |
| Pots ($m = n$) | $max\{k, n\}(2T(C) + 3T(A) + T(R)) + k(T(C) + T(A))$ |
| Big pot ($m = 1$) | $(n+1)T(C, \lceil \frac{k}{W} \rceil) + (n+2)T(A, \lceil \frac{k}{W} \rceil) + T(R)$ |
| Big pot ($m = n$) | $2nT(C, \lceil \frac{k}{W} \rceil) + 3nT(A, \lceil \frac{k}{W} \rceil) + nT(R)$ |
| $SMP(P_3)$ | $\frac{n}{2}(11T(M) + 13T(A) + 5T(C) + 2T(R))$ |
| $SMP(2K_2)$ | $\frac{n}{2}(9T(M) + 11T(A) + 7T(C) + 2T(R))$ |

Let us estimate the runtime functions in the following way $T(M) = A \cdot T(A) = R \cdot T(R)$, $T(C) = C \cdot T(M)$ and let $T(A, r) = r \cdot T(A), T(R, r) = r \cdot T(R), T(C, r) = r \cdot T(C)$ and let $W = 64$ (the number of bits in one number). This is a reasonable approximation on modern architectures and software.

Table 5: Comparing runtimes of algorithms in $T(M)$ with multiple parametrizations

| Parameters | | | | | |
|---|---|---|---|---|---|
| k | 30 | 365 | 365 | 365 | 100 |
| n | 30 | 30 | 30 | 30 | 1000 |
| C | 5 | 5 | 20 | 2 | 5 |
| A | 1/3 | 1/3 | 1/3 | 1/3 | 1/3 |
| R | 1 | 1 | 1 | 1 | 1 |
| **Method estimations** | | | | | |
| Voting based | 1089 | 3798 | 10458 | 2466 | 156078 |
| Pots ($m = 1$) | 520 | 6327 | 22752 | 3042 | 6528 |
| Pots ($m = n$) | 520 | 6327 | 22752 | 3042 | 12533 |
| Big pot ($m = 1$) | 167 | 1000 | 3790 | 442 | 10680 |
| Big pot ($m = n$) | 360 | 2160 | 7560 | 1080 | 24000 |
| $SMP(P_3)$ | 595 | 595 | 1720 | 370 | 19833 |
| $SMP(2K_2)$ | 705 | 705 | 2280 | 390 | 23500 |

The $SMP(P_3)$ is worse than $SMP(2K_2)$ only if $1 + A > C$, which is a really unlikely case. Clearly the pitfall of the pot algorithms is the big $k$ value.

When $k$ and $n$ are small, the big pot seems the most reasonable choice, but as $k$ gets bigger, it becomes infeasible. The graph-based approaches have strong restrictions $4 \mid n \vee 4 \mid n-1$. It can be seen in Table 5 that the simple pots algorithm

becomes the best when $n$ is large but $k$ remains small.

In future work we plan to create a scheme based on multiple different graphs to avoid restrictions and achieve the best performance at the same time.

# 4 Acknowledgement

Péter Hudoba was supported by EFOP-3.6.3-VEKOP-16-2017-00001: Talent Management in Autonomous Vehicle Control Technologies — The Project is supported by the Hungarian Government and co-financed by the European Social Fund.

Péter Burcsi has been supported by the European Union, co-financed by the European Social Fund (EFOP-3.6.2-16-2017-00013, Thematic Fundamental Research Collaborations Grounding Innovation in Informatics and Infocommunications).

# References

[1] Abramson, Morton and Moser, WOJ. More birthday surprises. *The American Mathematical Monthly*, 77(8):856–858, 1970.

[2] Alon, Noga, Caro, Yair, and Yuster, Raphael. Covering the edges of a graph by a prescribed tree with minimum overlap. *journal of combinatorial theory, Series B*, 71(2):144–161, 1997.

[3] Bárász, Mihály, Ligeti, Péter, Lója, Krisztina, Mérai, László, and Nagy, Dániel A. Another twist in the dining cryptographersprotocol. *Tatra Mountains Mathematical Publications*, 57(1):85–99, 2013.

[4] Bondy, John Adrian, Murty, Uppaluri Siva Ramachandra, et al. *Graph theory with applications*, volume 290. Citeseer, 1976.

[5] Boudot, Fabrice, Schoenmakers, Berry, and Traore, Jacques. A fair and efficient solution to the socialist millionaires problem. *Discrete Applied Mathematics*, 111(1-2):23–36, 2001.

[6] Chaum, David. The dining cryptographers problem: Unconditional sender and recipient untraceability. *Journal of cryptology*, 1(1):65–75, 1988.

[7] Dor, Dorit and Tarsi, Michael. Graph decomposition is npc-a complete proof of holyer's conjecture. In *Proceedings of the twenty-fourth annual ACM symposium on Theory of computing*, pages 252–263. ACM, 1992.

[8] Ergun, Ozlem, Kuyzu, Gultekin, and Savelsbergh, Martin. The lane covering problem. *Manuscript*, 2003.

[9] Goldreich, Oded. *Foundations of Cryptography: Volume 2, Basic Applications.* Cambridge University Press, New York, NY, USA, 2004.

[10] Hirt, Martin. *Multi Party Computation: Efficient Protocols, General Adversaries, and Voting*. Hartung-Gorre, 2001.

[11] Holyer, Ian. The np-completeness of some edge-partition problems. *SIAM Journal on Computing*, 10(4):713–717, 1981.

[12] Jukna, Stasys and Kulikov, Alexander S. On covering graphs by complete bipartite subgraphs. *Discrete Mathematics*, 309(10):3399–3403, 2009.

[13] Lin, Hsiao-Ying and Tzeng, Wen-Guey. An efficient solution to the millionaires problem based on homomorphic encryption. In *International Conference on Applied Cryptography and Network Security*, pages 456–466. Springer, 2005.

[14] Mathis, Frank H. A generalized birthday problem. *SIAM Review*, 33(2):265–270, 1991.

[15] Maurer, Ueli. Secure multi-party computation made simple. *Discrete Applied Mathematics*, 154(2):370–381, 2006.

[16] Pinkas, Benny. Fair secure two-party computation. In *International Conference on the Theory and Applications of Cryptographic Techniques*, pages 87–105. Springer, 2003.

[17] Shyu, Tay-Woei. Decomposition of complete graphs into paths and stars. *Discrete Mathematics*, 310(15-16):2164–2169, 2010.

[18] Wagner, David. A generalized birthday problem. In *Annual International Cryptology Conference*, pages 288–304. Springer, 2002.

[19] Yao, Andrew C. Protocols for secure computations. In *Foundations of Computer Science, 1982. SFCS'08. 23rd Annual Symposium on*, pages 160–164. IEEE, 1982.

[20] Yao, Andrew Chi-Chih. How to generate and exchange secrets. In *Foundations of Computer Science, 1986., 27th Annual Symposium on*, pages 162–167. IEEE, 1986.

Acta Cybernetica **24** (2019) 43–60.

# Benchmarking Graph Database Backends — What Works Well with Wikidata?

Tibor Kovács, Gábor Simon, and Gergely Mezei

**Abstract**

Knowledge bases often utilize graphs as logical model. RDF-based knowledge bases (KB) are prime examples, as RDF (Resource Description Framework) uses graph as logical model. Graph databases are an emerging breed of NoSQL-type databases, offering graph operations to process and manipulate data. Although there are specialized databases, the so-called triple stores, for storing RDF data, graph databases can also be promising candidates for storing knowledge. In this paper, we benchmark different graph database implementations loaded with Wikidata, a real-life, large-scale knowledge base. Graph databases come in all shapes and sizes, offer different APIs and graph models. Hence we used a measurement system, that can abstract away the API differences. For the modeling aspect, we made measurements with different graph encodings previously suggested in the literature, in order to observe the impact of the encoding aspect on the overall performance.

**Keywords:** graph database, knowledge base, Wikidata, benchmark

## 1 Introduction

Representing knowledge as a graph seems to be a natural choice from several aspects. People even without any specialized technical or natural science knowledge often organize concepts and relations between the concepts as nodes connected by edges. Some knowledge representation techniques also embraced this abstraction: RDF [21] represents metadata as a graph. Even the concept of knowledge graph has been floating around in recent years, without a clear definition [23]. We use this concept aligned with [26]: an RDF graph encoding a set of knowledge. A set of standards and technologies are built around the RDF concept. The so-called triple stores [15] emerged, a form of storage engines optimized to store a massive amount of RDF-modelled data. SPARQL standard [27] was also introduced as a query language to roam the RDF graphs.

In the DBMS world, graph as a data model is used since the dawn of database systems. As the NoSQL movement gained traction and as problem spaces with large-scale highly interconnected schemas—such as network simulation and social networks—demanded, a new family of NoSQL databases emerged, replacing the

DOI: 10.14232/actacyb.24.1.2019.5

key-value and the document concepts with graphs. The landscape of NoSQL graph databases (GDBs) is in flux even today, with various graph models, e.g., property graphs, hypergraphs, RDF graphs [42, 19], without standardized APIs, and even without a clear definition of a native graph database [41]. In our research, we focused on GDBs offering property graph model through Apache Tinkerpop API [13], a widespread property graph framework.

While connecting the dots above, storing knowledge represented as a graph in a database specialized to store graphs also seems a natural choice. However, one has to choose a graph database implementation first, that in turn determines the graph model and the API. Another decisive aspect is the graph encoding method. The RDF model gives a straightforward encoding for basic knowledge structures, however, there are different encoding models for reification [29], i.e., statements about statements. Reification is extensively used in KBs with reference management, where every statement should be backed up by external sources.

In order to help with these decisions, we selected a few graph database implementations and loaded with the same real-life, large-scale dataset, then queried with the same set of queries randomly generated from predefined query patterns. We run different measurements with different reification strategies. From the timing result of the query runs, we were able to construct the performance profile of each database—encoding strategy combination.

Our research aims to determine the performance characteristics of utilizing graph databases in various problem spaces. For the field of KBs, in the early phase, we worked with an algorithm-generated graph. Our initial results [34] showed counter-intuitive performance trends where more selective queries run slower than queries with more unbound values. In [30] the authors also encountered similar phenomena with a real-life dataset.

In this phase of our research, we also used Wikidata data, but we chose the databases exclusively from the family of NoSQL graph databases.

In this paper, we review the most important results connected to the research area. In Section Related Work, we present other's work related to this paper: benchmarks using Wikidata in which graph databases are involved and possible modeling solutions to the problem of reification. In Section Background, we describe the relevant part of the previous phase of the research: we give a short description of the already existing measurement system and how it is used to measure the performance of different DBMSs. After that, we give a detailed description of the measurement process in Section Experimental Settings: we introduce the dataset we used in the measurements, define the unified workflow of the benchmarking process, introduce the investigated database implementations, reification models and query patterns, and present the physical infrastructure on which the benchmarks were executed. Then we describe and analyze the results we got from the measurements in the Results section. Finally, we summarize our work, make some conclusions based on the results and present some of our plans for further enhancements.

## 2 Related Work

As performance is a key factor in the field of databases, several benchmarks have been conducted on graph databases. These measurements usually differ in the dataset used, in the query workloads, and in the benchmarked systems. In [32] several GDBs were loaded with the same generated graph and evaluated using a workload of loading, primitive graph operations, and traversals. Social networking is one of the primary problem spaces for GDBs. In [20] Angles et al. generated a synthetic graph with similar characteristics as a real-life social network, then executed a workload typical to this problem space (common friends, path search, etc.) on selected graph databases, triple stores, and relational engines. They have found that graph databases are more scalable in compute intensive graph problems than the concurrents.

The Linked Data Benchmark Council (LDBC)[9] is an independent authority "responsible for specifying benchmarks [...] for software systems designed to manage graph and RDF data." LDBC is continuously widening its benchmark portfolio: it has a framework for graph analytic tasks (breadth-first search, page rank, etc.)[31], social networking [24] and linked data (RDF)[33]. In [38] the authors run the LDBC social network benchmark against graph databases, triple stores and relational engines. They have found that more mature systems with heavily optimized query execution pipelines have the advantage over the more innovative newcomers—regardless of the database model type.

Meanwhile, the Linked Data community is looking for efficient storage solutions for RDF data. The LDBC's Semantic Publishing Benchmark [33] offers a measurement specification for comparing the performance of RDF engines. Recently, Pan et al.[39] surveyed the contemporary RDF benchmarks and management solutions. Moreover, the authors run the benchmarks against distributed RDF systems. In the end, they could not announce a clear winner, the performance depended heavily on the type of the query workload.

One of the key aspects of the benchmark dataset, that whether is it synthetic or real-life. Although synthetic datasets are trying to mimic some characteristics of a real-life dataset, Duan et al.[22] pointed out that benchmark datasets are tend to differ significantly in performance impacting metrics. In [35] Morsey et al. proposed a benchmark dataset and workload based on a real-life knowledge base. They also concluded that measurement results of a real-world dataset can be substantially different from the results of a synthetic dataset.

In [29] Hernández et al. compared the performance of several triple store databases on the same reified KB dataset. Later, as a follow-up, also Hernández et al.[30] compared the performance of DBMS's with different data models. They evaluated databases from different families, including relational, graph, triplestore, and used the publicly available and collaboratively edited knowledge base Wikidata [21] as the dataset. Due to the diverse data models, they had to use various encoding strategies for different database implementations. In [34] we loaded several graph databases with the same generated reified dataset, i.e., with an abstract, artificial knowledge base. As a natural next step, in the current phase of our research,

we replaced the generated data with a real-life knowledge base.

# 3 Background

## Modeling reification

The quasi-standardized way of reification was introduced in the early stages of the RDF specification [12]. It introduces a special vocabulary and a new node for every statement. The parts of the original statements are connected to this node with separate statements through to meta-predicates (rdf:subject, rdf:property, rdf:object) of the special vocabulary. Then, the meta-statements can use the intermediate node as the subject. We will be referring to this approach as standard reification. Standard reification is considered cumbersome and unnecessarily verbose. A somewhat leaner approach is proposed by implementing n-ary relations over the RDF model in [37]. Similarly, an intermediate node is introduced, connecting the object as well as other claim metadata to the subject. Hartig et al.[28] introduced an extension to the original RDF notation called RDF* by enabling using a whole statement as the subject, resulting a much shorter and clearer notation (Figure 2). Other reification modes like n-ary [25], singleton property [36] and named graph [29] were also proposed in the literature.

Graph databases usually offer more elaborate graph models than the basic RDF graph model. It seems promising that one can take advantage of these advanced constructs throughout the modeling of the reification. In [30] the authors mapped reified data to edge properties of the property graph model. At load time it worked, but typical queries involved edge properties had such a poor support, that they dropped this model. As a fallback, a form of standard reification model was implemented.

## Previous work

In [34], we created an easy-to-extend system for benchmarking graph DBMSs that we enhanced in the next phase of the research. The framework can be structured into several layers: the data source layer, the 1st conversion layer, the intermediate representation layer, the 2nd conversion layer, and the concrete implementation layer.

The data source layer is only responsible for providing the dataset for the measurement system so that it can be any kind of information source, like a Wikidata JSON dump or the output of a generator tool. As the experimental dataset is quite large, the loading process can be done efficiently using the DBMSs' bulk loader tools. As every inspected importer tool has a different input format, we defined an intermediate format so in case of $n$ different input type and $m$ different DBMS, one had to implement only $n+m$ converters instead of $n*m$, which is one of the key factors in extensibility. The responsibility of the 1st conversion layer is to convert the dataset from the data source layer to the defined intermediate format while the 2nd conversion layer is responsible for converting the intermediate

format dataset to the format expected by the DBMS specific import tool. When the dataset is loaded to a system, it goes into the concrete implementation layer.

For this infrastructure, we defined a unified measurement workflow in [34] for every DBMS-model pair: (i) every data from previous measurement (if any) must be deleted (ii) the source data must be converted to the intermediate format (iii) the data in intermediate format must be converted to the concrete loader format (iv) the dataset must be loaded into the DBMS (v) the queries must be converted to the current language-reification model representation (vi) the queries must be executed, measured and the results must be collected.

# 4 Experimental Setting

## Dataset

As we wanted to compare our results with the ones in [30], we used the same Wikidata JSON dump from January 2016. This dataset holds a massive knowledge base with 67 million statements. Wikidata highly encourages to back up the statements with references, hence reification is used extensively. As of December 2018, Wikidata holds 1.48 billion references for 654 million statements [18].

## Workload

For the same comparability reasons, we similarly generated the so-called atomic-lookup queries as in [30]. This simple query generation technique is based on the atomic parts of a single reified statement: the three parts of the base statement, with the property and the object of the metastatement. We generate a query by for each statement part either fill it with a fixed value or define it as a variable to project. As a result, we get 32 different query patterns. One of these patterns is explained in Figure 1.

## Reification models

Throughout our research, we examined three different reification techniques: (i) the property graph representation which encodes the qualifiers as edge properties, (ii) the standard reification, and (iii) the n-ary relation models which introduce a new node per each statement. Figure 2 depicts these models.

As opposed to [30], we did measurements using the property graph model since we wanted to compare the level of support between different graph database implementations. Edge properties look like a straightforward way to encode the reified claims. However, a reification claim referring to a resource would be an edge between an edge and a node resulting in an invalid graph model. One has to encode this kind of reference as an edge property with simple literal value referencing the identifier of the resource. Additionally, reification claims can reference multiple objects, e.g., a statement can be backed up by multiple sources. In that case, the edge property value would be a collection of identifiers.

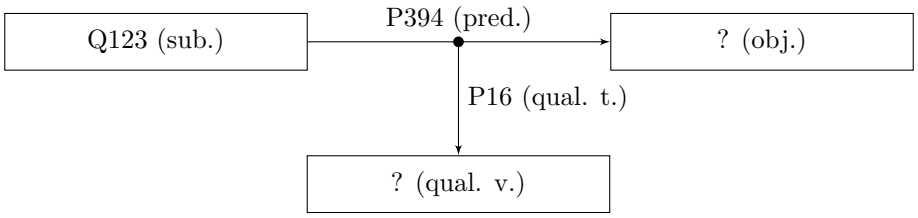



Figure 1: A reified statement has five parts: subject (sub.), predicate (pred.), object (obj.), qualifier type (qual. t.) and qualifier value (qual. v.), therefore the variability of this kind of statements can be described with five digit binary patterns, like 11010, where the first digit represents whether the subject part has a concrete value (1) or it is a variable (0), the second digit represents the same for the predicate part etc. in the previous exact order. This figure demonstrates the pattern 11010. The second and fourth numbers are 1, as they are bound to concrete values: the subject is a resource with id Q123, the predicate is P394, and the qualifier type is P16. The object and the qualifier value parts are variables. The example can be interpreted as follows: What values have the resource Q123 for property P394 and what values have these claims for property P16?

## Database implementations

In the current phase of the research, we selected three systems for the following reasons:

(i) We have chosen Blazegraph [1] database engine as a measurement subject, as its customized version currently serves [17] as a backend for Wikidata Query Service (WDQS)[16]. The primary model of Blazegraph is RDF, while SPARQL endpoint is offered for query purposes. These features make Blazegraph closely related to triple stores. However, the RDF model can be viewed and queried as property graph through Blazegraph's Apache Tinkerpop implementation [2].

(ii) The open source GDB called Titan [14] was the first pick for the WDQS backend role, but was dropped eventually due to governance changes and the high risk of abandonment. Later, the source code of Titan was forked, and JanusGraph [7] was born. This database implements almost all of its functionalities through the integration of other technologies; it supports various storage options [8] (e.g., BerkeleyDB, Apache Cassandra, Apache HBase) while utilizing Apache Tinkerpop as property graph engine.

(iii) Neo4j [40] was chosen, as it is currently considered one of the most popular graph database [3]. It is based upon the property graph model and supports the Tinkerpop stack and its Gremlin query language. Besides Gremlin, it defines its own declarative query language, called Cypher.

We did not examine every implementation-reification model pair for our experiments. We did not apply the property graph model on Blazegraph, as its primary model is RDF, it would map the edge properties (only available through

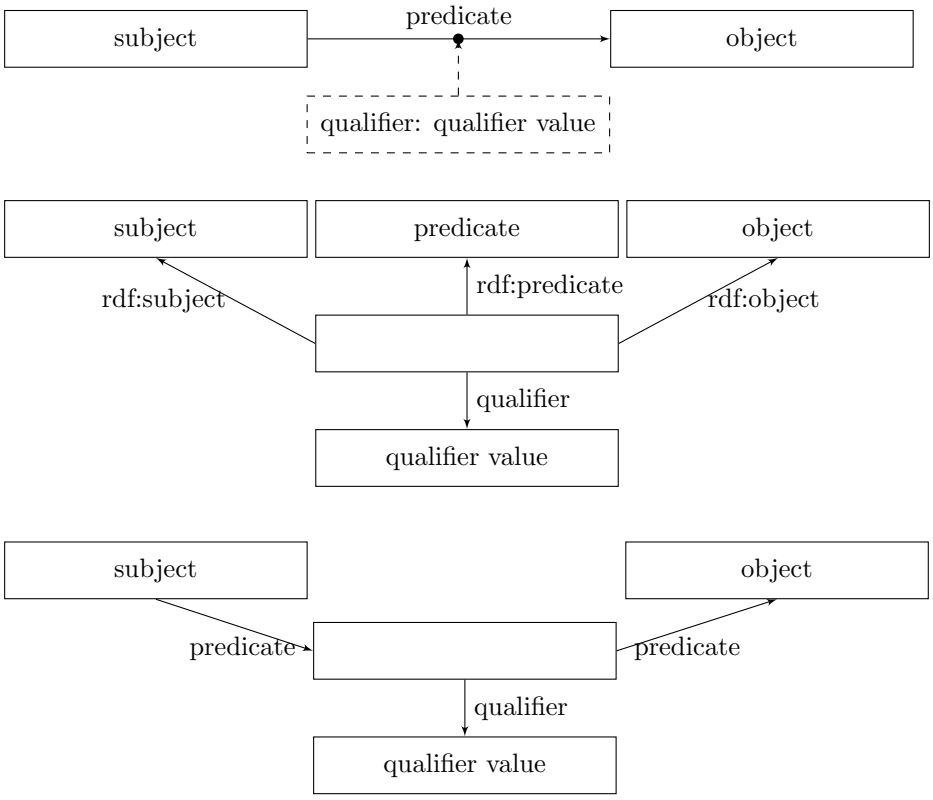



Figure 2: Visual representation of the different reification models. The first one is the property graph model, below that the standard model and finally the n-ary reification model.

the Tinkerpop interface) to RDF constructs. We did not utilize the RDF* notation support of Blazegraph. Moreover, the standard model was not measured on JanusGraph due to the extremely slow loading process. Table 1 summarizes the implementation-dependent measurement configurations.

We have investigated other GDBs as well, such as Grakn [4], OrientDB [10] and Gremlin API of Azure Cosmos DB [6], but we encountered a few difficulties during the modeling and loading phase. The primary cause of the problems was that these systems hardly support having multiple different values of the same property in a node or we did not find any available documentation on how to bulk load them into a database. OrientDB is a multi-model database; its earlier version was benchmarked as a GDB, e.g., in [32], but not with a real-life KB workload. Grakn and Cosmos DB's Gremlin are quite newcomers without any previous involvement in a significant benchmark effort in the academic literature.

Table 1: Overview of database configurations

| Impl. | Version | Model | Reification modes | Query language |
|---|---|---|---|---|
| Neo4j | 3.3.3 | Prop. graph | Prop. graph<br>Standard<br>N-ary | Cypher |
| Blazegraph | 2.1.4 | RDF (primary)<br>TinkerPop | Standard<br>N-ary | SPARQL |
| JanusGraph | 0.2.0 | TinkerPop | Prop. graph<br>N-ary | Gremlin |

In order to ensure the correctness of the results, the dataset is converted to the natively supported graph format of a system during the conversion processes, for example, we introduced edge and node properties in case of Neo4j and JanusGraph, RDF triples in case of Blazegraph. This means that every database worked with its native graph format, so the source of the dataset (a knowledge graph) is basically irrelevant.

## Installation environment

We provisioned separate virtual machine (VM) instances in the Azure public cloud for every GDB implementation. All VMs had a size of *Linux E4s v3*. They were configured with Intel XEON E5-2673 v4 processor containing 4 virtual CPU cores that support Intel Hyper-Threading Technology and 32 GiB of memory. A 64 GiB SSD was used as storage for the Ubuntu 16.04 LTS operating system and the particular DBMS. As the dataset had to be stored more times simultaneously—for example, during the conversions the source and the result dataset existed together at the same time—we added another SSD with 512 GiB capacity to store the dataset and the temporary files.

Besides the concrete DBMS implementation, we installed the Java and .NET Core runtime environments on all VMs. Even though the hosting environment is the same for the different DBMSs, the configuration of the systems can have a massive impact on their performance. As every investigated system is Java-based, we specified a uniform 20 GiB heap size for each system. On any other settings, we used their default configurations like [30] did.

## Measurement workflow

In this benchmark, we basically used the same measurement workflow described previously. The initial step was to delete all data that remained after the previous run. In the first phase, we transformed the decompressed JSON data to the import format of the concrete DBMS's import tool. After the transformation, we loaded the newly created dataset into the database. Finally, our tool inserted the previously random selected variable bindings into the query templates, measured the execution times and collected the mean query times. This workflow is depicted in Figure 3.

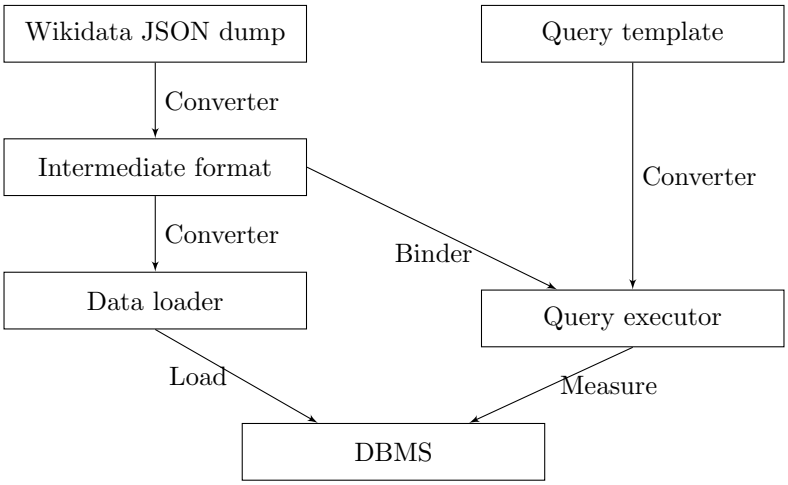



Figure 3: Overview of the measurement workflow

While in [34] we used a custom dataset generator to artificially create the dataset for our benchmarks, in this paper we replaced this component with a Wikidata JSON dump. For that reason, the 1st conversion layer had to be reimplemented as well, as the conversion between the data source and the intermediate representation is the responsibility of this layer.

In this paper, we present the measurements of a DBMS that we did not benchmark in [34], so we added a new component to the 2nd conversion layer that converts the dataset from intermediate representation to the format expected by the new DBMS's data loader tool. With the help of the intermediate representation layer, every other component could be reused in our new benchmark without nearly any modification.

Every query pattern (except the one without any variable) was run with ten different variable bindings. The $n$th query pattern ($q_n$) supplied with the $m$th variable binding for it ($b_{n,m}$) forms the runnable query $qb_{n,m}$. The values of the variables were randomly selected from the dataset in such a way that every query

would have a non-empty result set. To avoid first-time run transient phenomena, we ran all of the queries two times on every DBMS-encoding pair. In the beginning, we did not apply any time limit during the measurements, the first queries run by Neo4j were manually terminated after more than 15 minutes as the two runs would have taken more than a week continuous execution time based on our estimations. Afterwards, we set a query time limit to one minute, just like in [30] and in [34].

The workflow of the measurement phase consisted of the following steps (in this order): we applied the first bindings to the 32 query patterns $\{q_1, ...q_{32}\}$, then run the set runnable queries of $\{qb_{1,1}, qb_{2,1}, ...qb_{32,1}\}$, limiting each query separately to one minute. Then, we proceeded the same way with the remaining nine variable sets $\{qb_{1,2}, qb_{2,2}, ...qb_{32,2}, qb_{1,3}, ...qb_{1,10}, ...qb_{32,10}\}$. When the execution of all the runnable queries completed, the DBMSs were restarted to remove every memory content that could distort the results for the later runs. Finally, we made a second run by repeating the whole process with exactly the same pattern-binding combinations. The average response time values on the figures are calculated as the average of the twenty results for a given query pattern: the response times of the ten different variable bindings, i.e., the results from $\{qb_{i,1}, qb_{i,2}, ...qb_{i,10}\}$ for query pattern $i$—from the two separate runs. When a query had to be terminated because of the time limit, it was counted as 60 sec (actual time limit) in the average.

# 5 Results

Despite its popularity, Neo4j was the least performant system—in lots of cases by far compared to the other systems, as every query must have been terminated due to the time limit. We got these timeouts irrespective of the reification model. This experience is in line with [30] and [34].

We examined some of the query plans, and we found that the main reason for the poor performance is the lack of proper optimization. In some cases, we experienced that even though there were concrete nodes in the graph pattern to match, the pattern matching and the graph traversal started from variable nodes and the concrete information was used only in later phases. We investigated the usage of so-called planner hints [11]—adding explicit directions to the query optimizer into the query—but it resulted in performance improvements only in some cases. Furthermore, even the official documentation does not consider it as a general optimization technique [11].

The diagrams in Figure 4 show the results we got after measuring the performance of the Blazegraph. In contrast with Neo4j, most of the queries terminated before the time limit; only the most general patterns reached the one-minute timeout—the ones with only the qualifier part bound.

As can be seen in the diagram, there is no significant difference between the performance of the standard and the n-ary reification models. The results show that the two models do not just perform similarly, but they react almost the same way to the changes in the query patterns as it can be seen between 00100 and 00110 and in 01100.

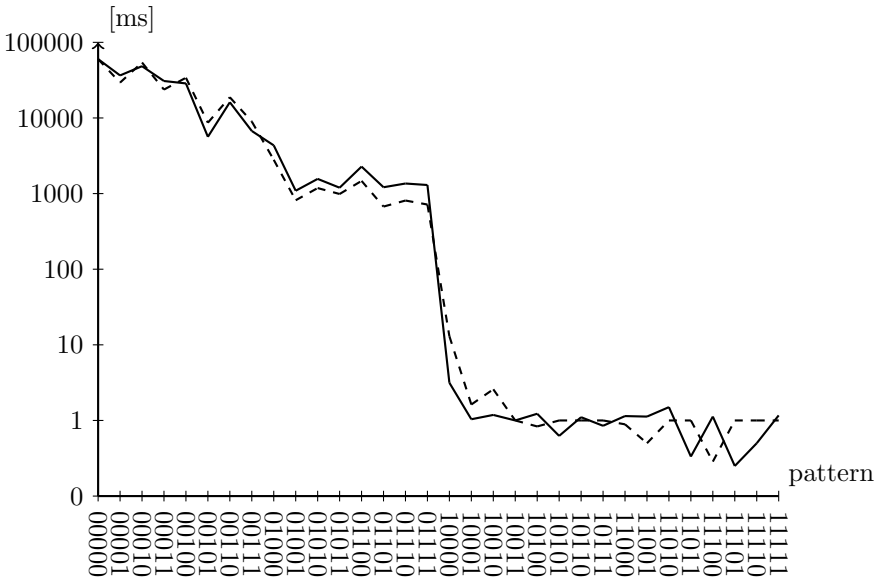



Figure 4: Blazegraph query mean response times for the standard (solid) and n-ary (dashed) models.

Even though the performance of the two models was quite similar, the n-ary model has an advantage against the standard model: it requires one less node to represent a statement, so on more massive datasets (like Wikidata), it requires significantly less storage space than the standard model.

Looking at the figure, it is quite conspicuous that there is a significant break in the middle of the graph. We have found that—in case of using Blazegraph—the most important factor that affects the elapsed time of an atomic-lookup is whether the subject (the starting point of the traversal) is concrete or not. Concretizing the subject means a significant performance boost, which suggests that Blazegraph pattern matching engine can perform reasonably well only if the graph traversal and the edge are pointing to the same direction.

One can see that the execution times continuously decreased before and after this gap as well. This constant performance improvement can be the result of the declarative nature of the SPARQL language, whose execution can be optimized using the up-to-date DB statistics in the more and more concrete queries.

Figure 5 shows the mean query response times of JanusGraph. One can see that there are fewer patterns in the diagrams, the ones ending with 01 are missing. That is because we encountered some difficulties in translating these queries into Gremlin queries. We tried using the proper Gremlin *Has* step variant in a couple of ways to filter the traversal according to the *01 patterns, but all of these queries

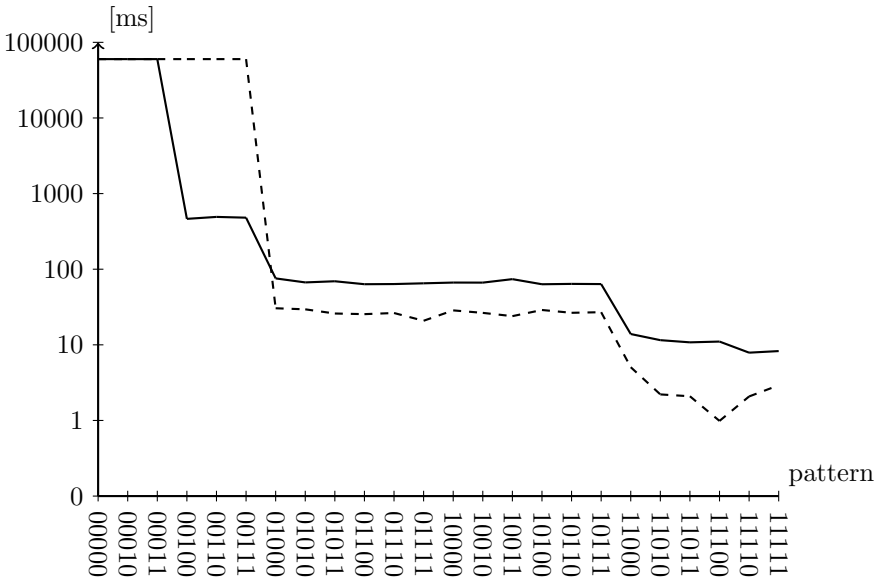



Figure 5: JanusGraph query mean response times for the n-ary (solid) and property graph (dashed) models.

returned with an empty result set. As we could not figure out why did this happen, eventually we removed these patterns from the measurement of JanusGraph.

This system also performs much better than Neo4j in most cases, but its query characteristic for atomic-lookups shows some similarities but also some fundamental differences with Blazegraph's. In case of both of these systems, the queries had to be terminated because of reaching the timeout only in the most general query patterns. While the n-ary model on JanusGraph timeouts on the patterns containing only qualifier information—like Blazegraph—, the queries with property graph model terminated before the time limit only when the query presented any type of concrete node information.

One can see that both systems have notable performance steps, but it emerges in a much more visible way in the case of JanusGraph. The n-ary model has a significant step at 00100 and two more small steps at 01000 and 11000. From that result, we concluded that the key factors that determine the performance of JanusGraph and n-ary model are the existence of concrete nodes but in contrast with Blazegraph, binding a concrete value to the predicate can result in much shorter response times. This DBMS-model pair can be ideal for queries containing edge information only, where it outperformed any other investigated system-model pairs.

The property graph model has a similar query characteristic to Blazegraph, as

it has its large performance step, where any kind of node information is presented in the query. In these cases, even though it performs significantly better than the n-ary model, it is much slower than Blazegraph. As the Blazergaph queries performed better for every query pattern than JanusGraph with property edge, we do not recommend using this pair in a real-life application.

Another interesting phenomenon is that the performance is almost constant between the steps on both models. This can be explained by the imperative kind of the Gremlin query language, as it gives a relatively small space to the optimizer to improve the query plans.


Figure 6: Comparing the results of the three investigated DBMSs in case of n-ary model: Neo4j (dotted), Blazegraph (dashed), JanusGraph (solid).

Based on the comparison of three measured systems on the only common reification model (Figure 6), one can come to the conclusion that Blazegraph offer the lowest response times if the subject part of the query is specified, otherwise JanusGraph outperforms all its competitors. Furthermore, neither of the systems could efficiently answer the questions that contain only qualifier information.

# 6 Conclusions and Future Work

In our work, we examined the performance of several graph database implementation — reification model pairs. We used a real-life knowledge base as dataset and simple atomic lookup queries as workload. Although the graph models offered by

GDBs seem rather suitable for knowledge graphs at first, one can hit quite a few limitations with datasets utilizing reification heavily. The direct, straightforward encoding of reified claims often resulted in a subpar performance as they relied on unoptimized features. In that manner, databases specialized to store knowledge models (e.g., RDF-based stores) and with the dedicated support of modeling reification clearly have advantage over general-purpose GDBs.

We concluded that the execution times depend heavily on both the query pattern and the system-encoding pair. The general tendency is that the less node variable a query has, the faster its execution is. Event though as the results show, the execution times slightly depend on the selected representation model, its impact is far less than the DBMS implementation used.

Based on the overall average query times measured, the best performance for this kind of workload can be reached by using Blazegraph with either n-ary or standard encoding. Considering other factors than performance, our choice would be Blazegraph with n-ary representation, as this representation has lower storage footprint.

As every benchmark, our work has its limitations. In the current phase, we had to apply several constraints, for example on the workflow, on the measured configurations or on the used query languages. Currently, we are working on relaxing these constraints. Our measurements were limited to atomic-lookup queries, but in the future, we plan to investigate the performance of the DBMS's on a more real-life workload. To achieve this, we are analyzing the most frequently used query patterns provided by the Wikidata query service. Once we get these statistics, we can compose a query set that can simulate nearly real-life questions. Using these queries, we can measure the performance of the implementation-model pairs on a realistic load, which will give a better view on when and how to use these systems.

In the current phase of our work, we analyzed the query execution results and some of the obtained query plans to find out why does a query run slowly while others are fast. In the future, we are planning to make some deeper analysis, even at the implementation level, as most of these systems are open-sources.

As we introduced the use of an intermediate representation in the conversion phase, the measurement system can be extended effortlessly. Thus, we are planning to involve other databases, like Grakn [4], Azure CosmosDB [6] or HypergraphDB [5]. We are also planning the investigate other (even system specific) reification techniques such as the general singleton property [29] approach or the RDF* mode of Blazegraph [28].

Furthermore, we are planning to introduce the query language as a new dimension in the future. Until now, we investigated only the "native" language of a DBMS, even though they usually support more than one, for example, Blazegraph supports Gremlin. We are planning to measure the same database with different languages (if possible) to determine how much influence does it have on the performance. This may answer a couple of open questions, like whether the declarativity of a language or the language itself has any impact on the performance of these systems.

## Acknowledgments

The project was funded by the European Union, cofinanced by the European Social Fund (EFOP-3.6.2-16-2017-00013). Cloud computing resources were provided by a Microsoft Azure for Research award.

## References

[1] Blazegraph products. https://web.archive.org/web/20171125161035/https://www.blazegraph.com/product/. Accessed: 2018-03-13.

[2] Blazegraph TinkerPop implementation. https://web.archive.org/web/20180611150556/https://github.com/blazegraph/tinkerpop3. Accessed: 2018-09-09.

[3] DB-Engines ranking of graph DBMS. https://web.archive.org/web/20180911002043/https://db-engines.com/en/ranking/graph+dbms. Accessed: 2018-03-13.

[4] Grakn.AI - the knowledge graph. https://web.archive.org/web20180918085112/http://www.grakn.ai/grakn-core. Accessed: 2018-08-02.

[5] HypergraphDB - a graph database. https://web.archive.org/web/20180809121925/http://hypergraphdb.org/. Accessed: 2018-08-02.

[6] Introduction to Azure Cosmos DB: Graph API. https://web.archive.org/web/20180911002034/https://docs.microsoft.com/en-us/azure/cosmos-db/graph-introduction. Accessed: 2018-08-02.

[7] JanusGraph. https://web.archive.org/web/20180919165133/http://janusgraph.org/. Accessed: 2018-03-13.

[8] JanusGraph storage backends. https://web.archive.org/web/20180209145536/http://docs.janusgraph.org:80/latest/storage-backends.html. Accessed: 2018-03-13.

[9] Linked Data Benchmark Council. https://web.archive.org/web/20181228154821/http://ldbcouncil.org/. Accessed: 2018-08-06.

[10] OrientDB. https://web.archive.org/web/20181016165245/https://orientdb.com. Accessed: 2018-12-07.

[11] Planner hints and the USING keyword. https://web.archive.org/web/20180206081105/http://neo4j.com:80/docs/developer-manual/current/cypher/query-tuning/using/. Accessed: 2018-08-03.

[12] RDF schema 1.1 - reification vocabulary. https://web.archive.org/web/20180920184035/https://www.w3.org/TR/rdf-schema/. Accessed: 2018-08-06.

[13] TinkerPop3 documentation. https://web.archive.org/web/20180923130832/http://tinkerpop.apache.org/docs/3.3.3/. Accessed: 2018-08-06.

[14] Titan Graph Database. https://web.archive.org/web/20180910214447/http://titan.thinkaurelius.com/. Accessed: 2018-09-09.

[15] What is RDF triplestore? https://web.archive.org/web/20170506152814/http://ontotext.com:80/knowledgehub/fundamentals/what-is-rdf-triplestore/. Accessed: 2018-09-11.

[16] Wikidata Query Service. https://query.wikidata.org/. Accessed: 2018-09-09.

[17] Wikidata Query Service - user manual. https://web.archive.org/web/20180917181601/https://www.mediawiki.org/wiki/Wikidata_Query_Service/User_Manual. Accessed: 2018-09-09.

[18] Wikidata statistics dashboard for references. https://grafana.wikimedia.org/d/000000182/wikidata-datamodel-references?orgId=1&from=1514836723618&to=1543694323619. Accessed: 2018-12-11.

[19] Angles, Renzo. A comparison of current graph database models. In *Proceedings of the 2012 IEEE 28th International Conference on Data Engineering Workshops*, ICDEW '12, pages 171–177, Washington, DC, USA, 2012. IEEE Computer Society. DOI: 10.1109/ICDEW.2012.31.

[20] Angles, Renzo, Prat-Pérez, Arnau, Dominguez-Sal, David, and Larriba-Pey, Josep-Lluis. Benchmarking database systems for social network applications. In *First International Workshop on Graph Data Management Experiences and Systems*, GRADES '13, pages 15:1–15:7, New York, NY, USA, 2013. ACM. DOI: 10.1145/2484425.2484440.

[21] Cyganiak, Richard, Wood, David, Lanthaler, Markus, Klyne, Graham, Carroll, Jeremy J, and McBride, Brian. RDF 1.1 concepts and abstract syntax. *W3C recommendation*, 25(02), 2014.

[22] Duan, Songyun, Kementsietsidis, Anastasios, Srinivas, Kavitha, and Udrea, Octavian. Apples and oranges: A comparison of RDF benchmarks and real RDF datasets. pages 145–156, 01 2011. DOI: 10.1145/1989323.1989340.

[23] Ehrlinger, Lisa and Wöß, Wolfram. Towards a definition of knowledge graphs. In *SEMANTiCS*, 2016.

[24] Erling, Orri, Averbuch, Alex, Larriba-Pey, Josep, Chafi, Hassan, Gubichev, Andrey, Prat, Arnau, Pham, Minh-Duc, and Boncz, Peter. The LDBC social network benchmark: Interactive workload. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 619–630, New York, NY, USA, 2015. ACM. DOI: 10.1145/2723372.2742786.

[25] Erxleben, Fredo, Günther, Michael, Krötzsch, Markus, Mendez, Julian, and Vrandečić, Denny. Introducing Wikidata to the linked data web. In *Proceedings of the 13th International Semantic Web Conference (ISWC'14)*, volume 8796 of *LNCS*, pages 50–65. Springer, 2014.

[26] Färber, Michael, Bartscherer, Frederic, Menne, Carsten, and Rettinger, Achim. Linked data quality of DBpedia, Freebase, OpenCyc, Wikidata, and Yago. *Semantic Web*, pages 1–53, 2016.

[27] Harris, Steve, Seaborne, Andy, and Prudhommeaux, Eric. SPARQL 1.1 query language. *W3C recommendation*, 21(10), 2013.

[28] Hartig, O. and Thompson, B. Foundations of an Alternative Approach to Reification in RDF. *ArXiv e-prints*, June 2014.

[29] Hernández, Daniel, Hogan, Aidan, and Krötzsch, Markus. Reifying RDF: what works well with Wikidata? In *Proceedings of the 11th International Workshop on Scalable Semantic Web Knowledge Base Systems (SSWS 2015)*, volume 1457 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2015.

[30] Hernández, Daniel, Hogan, Aidan, Riveros, Cristian, Rojas, Carlos, and Zerega, Enzo. Querying wikidata: Comparing SPARQL, relational and graph databases. In *International Semantic Web Conference*, pages 88–103. Springer, 2016.

[31] Iosup, Alexandru, Hegeman, Tim, Ngai, Wing Lung, Heldens, Stijn, Prat-Pérez, Arnau, Manhardto, Thomas, Chafio, Hassan, Capotă, Mihai, Sundaram, Narayanan, Anderson, Michael, Tănase, Ilie Gabriel, Xia, Yinglong, Nai, Lifeng, and Boncz, Peter. LDBC graphalytics: A benchmark for large-scale graph analysis on parallel and distributed platforms. *Proc. VLDB Endow.*, 9(13):1317–1328, September 2016. DOI: 10.14778/3007263.3007270.

[32] Jouili, S. and Vansteenberghe, V. An empirical comparison of graph databases. In *2013 International Conference on Social Computing*, pages 708–715, Sept 2013. DOI: 10.1109/SocialCom.2013.106.

[33] Kotsev, Venelin, Minadakis, Nikos, Papakonstantinou, Vassilis, Erling, Orri, Fundulaki, Irini, and Kiryakov, Atanas. Benchmarking RDF query engines: The LDBC semantic publishing benchmark. In *BLINK@ ISWC*, 2016.

[34] Kovács, Tibor. Nagyméretű szemantikus adathalmazok tárolási megoldásainak teljesítményközpontú összehasonlítása. In *BME-VIK TDK*, 2017.

[35] Morsey, Mohamed, Lehmann, Jens, Auer, Sören, and Ngonga Ngomo, Axel-Cyrille. Dbpedia SPARQL benchmark – performance assessment with real queries on real data. In Aroyo, Lora, Welty, Chris, Alani, Harith, Taylor, Jamie, Bernstein, Abraham, Kagal, Lalana, Noy, Natasha, and Blomqvist, Eva, editors, *The Semantic Web – ISWC 2011*, pages 454–469, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[36] Nguyen, Vinh, Bodenreider, Olivier, and Sheth, Amit. Dont like RDF reification? *Proceedings of the 23rd international conference on World wide web - WWW 14*, page 759770, Apr 2014. DOI: 10.1145/2566486.2567973.

[37] Noy, Natasha, Rector, Alan, Hayes, Pat, and Welty, Chris. Defining n-ary relations on the semantic web. *W3C working group note*, 12(4), 2006.

[38] Pacaci, Anil, Zhou, Alice, Lin, Jimmy, and zsu, M. Tamer. Do we need specialized graph databases?: Benchmarking real-time social networking applications. pages 1–7, 05 2017. DOI: 10.1145/3078447.3078459.

[39] Pan, Zhengyu, Zhu, Tao, Liu, Hong, and Ning, Huansheng. A survey of RDF management technologies and benchmark datasets. *Journal of Ambient Intelligence and Humanized Computing*, 9(5):1693–1704, Oct 2018. DOI: 10.1007/s12652-018-0876-2.

[40] Robinson, Ian, Webber, Jim, and Eifrem, Emil. *Graph Databases*. OReilly Media, 2015.

[41] Rodriguez, Marko A. A letter regarding native graph databases. https://web.archive.org/web/20180828112004/https://www.datastax.com/dev/blog/a-letter-regarding-native-graph-databases, 2013.

[42] Rodriguez, Marko A. and Neubauer, Peter. Constructions from dots and lines. *CoRR*, abs/1006.2361, 2010.

Acta Cybernetica **24** (2019) 61–81.

# Keeping P4 Switches Fast and Fault-free through Automatic Verification*

Dániel Lukács[a], Gergely Pongrácz[b], and Máté Tejfel[a]

**Abstract**

The networking dataplane is going through a paradigm shift as softwarization of switches sees an increased pull from the market. Yet, software tooling to support development with these new technologies is still in its infancy. In this work, we introduce a framework for verifying performance requirement conformance of data plane protocols defined in the P4 language . We present a framework that transforms a P4 program in a versatile symbolic formula which can be utilized to answer various performance queries. We represented the system using denotational semantics and it can be easily extended with low-level target-dependent information. We demonstrate the operation of this system on a toy specification.

**Keywords:** P4, network verification, data plane, performance modeling, cost analysis

## 1 Introduction

Currently in the networking industry, network devices are being commoditized fast and software gets more and more market share as consumers want scalable and easily replaceable devices, while vendors want to keep development costs low. For example, software-defined networking (SDN) and Network function virtualization (NFV) technologies address this need by allowing network administrators to dynamically control network topology, configurations, and protocols.

New languages are emerging, aiming to assist network engineers to define the functioning of switches or network functions in the forwarding plane. Among them, P4 [7, 15] intends to keep the best aspects of both hardware and software by enabling network engineers to communicate their intent in a general high-level language, while the task of compiling high-level protocol description to low-level target

*The research has been supported by the European Union, co-financed by the European Social Fund (EFOP-3.6.2-16-2017-00013).

[a]Eötvös Loránd University, Faculty of Informatics, Dept. of Programming Languages and Compilers, E-mail: {dlukacs,matej}@caesar.elte.hu

[b]Ericsson Hungary, E-mail: Gergely.Pongracz@ericsson.com

DOI: 10.14232/actacyb.24.1.2019.6

architectures is delegated to the backend software. The hybrid approach is highly effective, but burdens backends, static analyzers, and verification frameworks, as now these also have to take into account low-level targets. We recommend [13] and [7] on the transpiring software revolution in networking, the growing empowerment of network operators at the expense of switch vendors, and the question whether P4 will free the network from fixed-function switching interfaces (e. g. OpenFlow) by making protocol-inpedendent packet processing possible.

In this work, we present the first iteration of a framework for verifying *functional requirements* and *non-functional requirements* of network protocols in P4. Our focus in this paper is checking whether P4 programs satisfy performance requirements using cost analysis methodologies. In Section 1.1, we show that in networked environments, conformance to performance requirements determines if the switch can serve its purpose or instead it will get overflowed with packets and introduce network-wide anomalies.

## 1.1 Motivation

The goal of cost analysis is to approximate the execution cost of a program using the program code, without executing the program itself. A cost analysis system that is capable of giving exact execution costs for any problem is also capable of solving the halting problem. This implies that we have to stay content with approximations.

While it is always nice to have an idea about the costs of executing the programs we develop, this issue is more pressing with packet processing programs (such as P4) running on networked software switches. In any network – with or without P4 –, the number of packets processed in a unit of time (or energy used) correlates strongly with the unit of service prodvided (or rate of profit produced) by the network. Specifically for P4, one of the big promises of the language is that switches executing P4 protocols can combine the generality of NVFs software switches with the speed of earlier SDN switches (such as OpenFlow) that were closer to hardware, but only supported a fixed amount of protocols.

Moreover, beyond "more is better" being a desirable non-functional requirement, an easily overlooked fact is that the performance of a switch program is actually an important factor in the functional correctness and usability of that program, similarly to real time systems. In short, unless the switch is performant enough to serve all incoming requests in time, the packets will start accumulating in the packet buffers (installed for load balancing the temporary increases in demand), and upon buffer overflow, packets start getting lost, producing unexpected network behaviors.

To demonstrate a realistic requirement, let us examine switches in a 10 Gigabit Ethernet network. In such a network, the maximum incoming throughput is $10 Gbps = 1.25 GBps = 1250 MBps$. Assuming somewhat pessimistically that only minimal Ethernet frames with no payload are transferred, the size of the packets will be the size of the Ethernet frames, that is $8B + 64B + 12B = 84B$. From this, the incoming packet rate measured in *Mpps* (Million Packets Per Second) is

$1250MBps/84 \approx 14.88Mpps$. This means that a latency of $1/14.88s \approx 0.067s$ is allowed for 1 million packets, which is a latency of $10^9 * 1/(14.88 * 10^6)ns \approx$ **67ns** for 1 packet. Assuming a modern CPU with $4.3GHz = 4.3$ $cycles/ns$ clock speed (such as the one alluded by Figure 13 in Section 4), we can conclude that at most $4.3$ $cycles/ns \cdot 67ns \approx$ **288 cycles** can be spent for processing a packet to stay safe from buffer overflows.

One observation regarding this calculation can be that performance requirements towards switches turn out be quite strict. Another observation is that verification of such small boundaries will inherently require factoring in machine-level operations, such as the execution costs of various CPU instructions and accessing caches and main memory.

Yet another promising feature of P4 over earlier NFV and SDN approaches is that it is a well-designed high-level programming language with standardised syntax and semantics, enabling formal, well-generalisable analysis of switch and network behavior.

In this work, we present a formal system capable of taking into account the aforementioned low-level operations and statically deriving strict cost estimates from a represenation of P4 program semantics.

## 1.2 About P4

P4 programs describe the control flow of packet processing network devices, commonly called switches. One peculiarity of P4 is that certain control structures are intentionally left unspecified by the designers. Implementation questions are left to the compilers targeting different platforms: this way compiler designers can choose the most efficient solutions for their target platform. Moreover, the lack of superflous restrictions will enable more platforms to adopt the language. The price of this feature is that P4 programs cannot be generally analyzed without sufficient, low-level knowledge about the selected platforms. One of our goals in this work was to design a system that analyzes and verifies P4 independently of any platforms as deep as possible, and can be easily extended with platform specific information to achieve completeness.

Figure 1 depicts a P4 program. Here, the call to `V1Switch` lists the arguments (similar to function pointers) of a P4 pipeline. The implementation of `V1Switch` is unspecified, but from the interface description we can find out that incoming packets will be first processed by a parser, called `ParserImpl`. After the parsing phase, `V1Switch` will continue with following phases, each operating on the data structure containing the parsed packet (`headers`).

The parser is defined in P4, using state transitions. The parser control flow is illustrated by the state machine diagram in Figure 2 (side effects were abstracted away). Starting from state `start`, state `parse_eth` is immediately reached, and then, if the packet header signifies that the packet is an IPv4 packet (field `ethType` equals to 2048 in decimal), the next state is `parse_ipv4`. In both cases, the state machine goes in the accepting state, `accept`. Whether a packet is an IPv4 packet or not only makes a difference in side effects.

```
1  // "basic_routing-bmv2.p4"
2  V1Switch(ParserImpl(),
3           ingress (),
4           verifyChecksum(),
5           egress (),
6           computeChecksum(),
7           DeparserImpl()) main;
8
9  parser ParserImpl(packet_in packet,
10                   out headers hdr,
11                   inout metadata meta,
12                   inout standard_metadata_t stmeta){
13
14   state start {
15     transition parse_eth;
16   }
17
18   state parse_eth {
19     packet.extract(hdr = hdr.eth);
20
21     transition select(hdr.eth.ethType) {
22       16w0x800: parse_ipv4;
23       default: accept;
24     }
25   }
26
27   state parse_ipv4 {
28     packet.extract(hdr = hdr.ipv4);
29     transition accept;
30   }
31 }
32  ....
```

```
34  ....
35
36 header ethernet_t {
37   bit<48> dstAddr;
38   bit<48> srcAddr;
39   bit<16> ethType;
40 }
41
42 header ipv4_t { ... }
43
44 struct headers {
45   ethernet_t eth;
46   ipv4_t     ipv4;
47 }
48
49  ....
50
51 // "core.p4"
52 extern packet_in {
53
54   void extract<T>(
55     out T hdr);
56
57   void advance(
58     in bit<32> size);
59
60   bit<32> length();
61 }
62
63 // "v1model.p4"
64 ...
65
```

Figure 1: Excerpt of a P4 program.

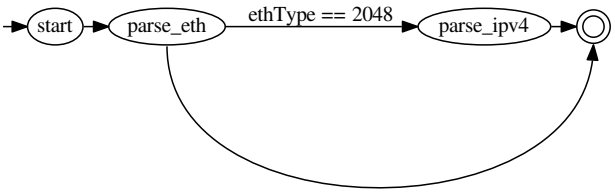



Figure 2: An automaton illustrating the parser in Figure 1.

In certain states the parser reads parts of the packet into the P4 program memory space. The memory space storing the packet, and the related method reading from this storage is declared by the data structure called `packet_in`. *Extern* data structures – such as `packet_in` – and methods are also unspecified. On the other hand, data structures such as `headers`, and `ethernet_t` are completely defined by the P4 code, which means we can work with these inside most P4 function constructs. For brevity, we included only the parts related to the parser that we analyze in this work. We believe the procedure intrduced in this paper can be effectively generalized for other control structures in the language – such as match-action tables – but we deem the validation of this claim as future research topic.

### 1.3 Contributions

Earlier, we enumerated our current research goals and some of the related analysis problems posed by P4. In this section, we intend to highlight specifically those problems we address in the current paper. We also showed earlier that in networked environments it is critical for switches to conform to specific time requirements, otherwise they cannot reliably provide the expected functionality. In this work, we outline a system that, given switch programs in a subset of P4 and adequate platform specifications, infers performance information that can be utilized to automatically verify whether given program satisifes given performance requirements on the given platform (see Section 2). This language subset was selected in hope that it covers a wide-enough range of challenges (such as target-dependence and low-level semantics) posed by P4, so that our system can be extended for the whole language.

For analyzing the cost of P4 programs, we adapted cost analysis approaches in existing literature [3, 16, 5] to P4. Our approach utilizes program transformations over formal representations of P4 program semantics (see Section 3). Advantages of the denotational approach is that it enables formal reasoning about its correctness, and its compositionality makes it easy to plug in target-specific information. Disadvantages are mostly related to efficiency: to increase precision we need to lower the level of abstraction we work on, and we can expect the amount of information on each abstraction level to grow exponentially (e. g. interfacing with the NIC, memory access, caching, cost of CPU instructions must all be carefully considered). To keep the rules simple, we utilize A-Normal form [9] that immensely simplifies function call evaluation semantics. Size and time efficiency of cost formula evaluation is assured by the introduction of let expressions (or alternatively nested lambda expressions) that can be used to eliminate redundant expressions and memoize intermediate results.

Various queries answering various performance questions can be created simply by parameterizing the symbolic formula resulting from the above process. As giving estimations with industrial-level precision for one or more platforms is out of the scope of this paper, we demonstrate the operation and application of the presented system using a toy specification of a fictional target in Section 4.

Our reference implementation is realized as a backend for the P4C compiler [1] and it heavily utilizes the Pure term rewriting system [6].

## 2 Cost analysis framework for P4

A challenge specific to P4 (although also occurring in other languages, such as C) is the handling of terms undefined by the specification, hereinafter referred to as *unspecified* terms to avoid confusion with theoretically undefined terms (such as division by zero, and the value of infinite recursion). It is the job of the compiler, to link unspecified structures – such as the extern object `packet_in` or the pipeline `V1Switch` in Figure 1 – to definitions that can be executed efficiently on the platform. Figure 3 depicts the data flow model we defined to address the challenge of

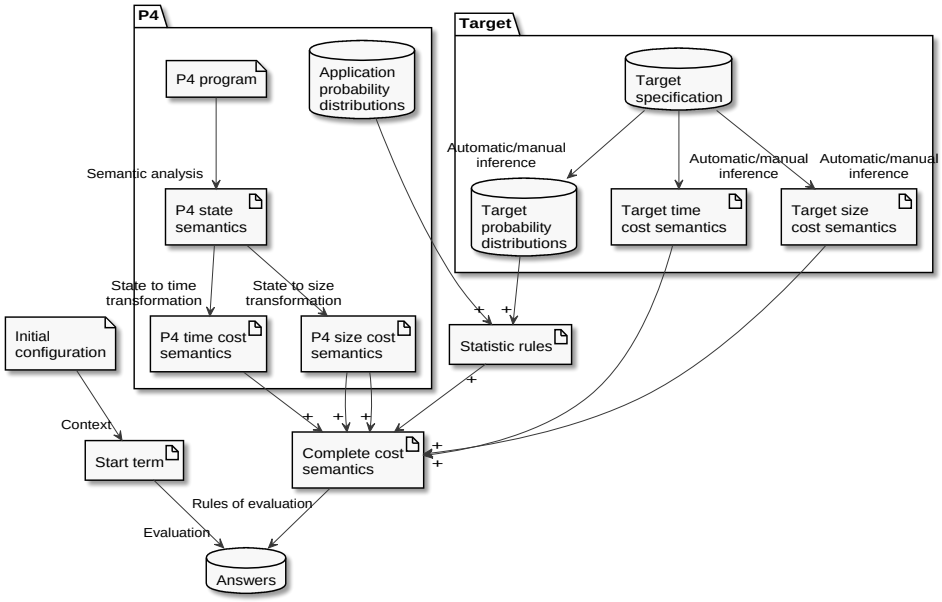



Figure 3: The complete architecture of the cost analysis framework

analyzing and verifying target-dependent P4 code.

The P4 program is first parsed into an intermediate representation (IR) called *state semantics*, maybe utilizing IRs in existing P4 compiler solutions, such as P4C [1] or T4P4S [1]. This represents the program as mapping between explicit memory states before and after program execution. This representation is then transformed to two kinds of abstract semantics used specifically to analyze execution cost. One we will refer to as *time semantics*, as it maps meaningful size abstractions of input states to a numeric value characterizing program execution cost, such as CPU cycles. The other we will refer to as *size semantics*, as it maps meaningful size abstractions of input states to meaningful size abstractions of output states. We detail these representations and the transformation between them in Section 3.

As these transformations depend only on the P4 code, they are insufficient in themselves for completely deriving the cost of P4 programs, as this requires target-specific information. Our system expects this in the form of *target-specific time semantics and size semantics* rules. These rules either have to be delivered manually by developers employed by the target vendor, or it may be possible to automatically infer them, given a sufficiently formal specification describing the behavior of the target. We should note that such automatic inference from arbitrary target-dependent state semantics (or some other representation) requires further research efforts as it may introduce unexpected problems: on the target level we lose the comfortable guarantees provided by P4, such as upper-bounded loops or no

loops at all, compile-time known memory sizes, structured control flow expressions and high-level data structures.

As it is common in static analysis, considering all possible program states (or even program inputs) would be unfeasible, and through abstraction we can acquire feasibility by trading away precision. In cost analysis, the problem manifests itself when we are dealing with conditional control flow. As predicates cannot be evaluated without the input, we either have to represent the costs of conditional execution paths as dependent on an unevaluated predicate, or – by further abstraction – we can treat the predicate as a random variable and apply statistical functions to these costs to produce performance information that is imprecise but useful in practice.

Figure 3 also includes these required statistical functions and informations in the architecture. As some of the predefined statistics (such as the average cost) require knowledge about the probability distribution of the predicates, this information is also required to perform the analysis. The two kinds of probability distributions in the figure relate to a distinction between predicates appearing in the semantics: some predicates are introduced in the target semantics (such as checking for cache hits), while others are dependent on program input and program context (such as deciding whether a given header can be parsed from a packet in a given parser state, or whether a packet matches any entry in the match-action table).

Given all these informations and a target-dependent abstract initial program memory state, we can finally evaluate the abstract call to the program entry point with any or all of the predefined statistics to acquire performance information about the P4 program and verify whether or in what circumstances does it conforms to the performance requirements.

In Section 4, we also go through the most important steps of this process with illustrations.

## 3 Transforming programs to program costs

In this section, we present the program transformation system used to derive time and size semantics from the state semantics of a P4 program. We realized the transformation as a term rewriting system containing reduction rules, analogous to function definitions in the lambda calculus. An advantage of functional style beyond formality is that it automatically guarantees confluence as per the Church-Rosser theorem.

For the ease of reproducibility, the examples in this paper were formalized in the executable term rewriting language, Pure [6]. Pure mostly follows the notational naming conventions of ML-style functional languages. We give basic description for less familiar syntax elements in Pure, but ultimately we have to refer the reader to the Pure language manual. To separate the meta-syntax (i. e. the syntax of Pure) from the concrete syntax of the semantics (defined by EBNF grammars in this paper), we typeset meta-syntactical symbols in bold, meta-syntactical variables using normal fonts, and typeset all concrete symbolic values in italic. In Figure 4

```
⟨App⟩          ::= ⟨Name⟩ ⟨State⟩ ⟨Scope⟩

⟨TimeCost⟩     ::= 'TIME' ⟨Expr⟩ ⟨SizeCost⟩ ⟨Scope⟩
                 | ⟨TimeCost⟩ '+' ⟨TimeCost⟩
                 | ⟨ProbDist⟩ '*' ⟨TimeCost⟩
                 | ⟨Number⟩ '*' ⟨TimeCost⟩
                 | ...

⟨SizeCost⟩     ::= 'SIZE' ⟨Expr⟩ ⟨SizeCost⟩ ⟨Scope⟩
                 | ...

⟨Reference⟩    ::= ⟨State⟩ ['◇' ⟨Scope⟩] {'!' ⟨Name⟩}

⟨RndReference⟩ ::= '𝒳' ['◇' ⟨Scope⟩] {'!' ⟨Name⟩}

⟨ProbDist⟩     ::= '𝒫' ⟨Predicate⟩
```

Figure 4: EBNF syntax for the most frequent expressions used in this paper.

we find mixed rules from the EBNF noindent grammars describing the languages we are using for expressing state, time and size semantics of P4 programs.

Function applications in the state semantics apply the definition referred by the given name, to the argument which is a program state. Note that the grammar enforces A-normal form (ANF) [9]: applications are only allowed to have variable symbols and concrete states as arguments. The state semantics syntax exclusively uses lexical scoping (instead of the mixed lexical-dynamical approach of P4): a mapping of names are passed to called functions. The names in the scope are used in function definitions to resolve references pointing to the state. The exclamation mark (`!`) is Pure syntax for record field access while ◇ was defined by us to handle sequences of field accesses, since `!` is left-associative in Pure. We represent concrete (i. e. transformable) *let expressions* and *case analyses* in Pure's built-in syntax (with `_when_` and `_case_` respectively).

Time and size costs of function definitions are denoted with the `TIME` and `SIZE` expressions: these work similarly to applications, but they evaluate to time and size costs instead of program states. We also include expressions required for probabilistic handling of case analyses, and a few target-defined constants appearing in Section 4.

Next, we present the rules §3.1–§3.4 forming the transformation system (a meta term rewriting system) between state semantics and time semantics. We expect that only only one system is loaded in the rewriting environment at a time, so the only subexpressions expanded are those appearing on the left sides. We will assume, that term rewriting rules – represented here with an arrow ( $\stackrel{\cdot}{\Longrightarrow}$ ) between the two sides – are also part of the concrete syntax: they can be created, transformed, and added to programs during runtime.

Rules §3.1 and §3.2 apply the time cost function to both sides of a state rule (similarly to how we usually do this to equations). The former one is an exception for program entry, mapping a concrete input state to its size concrete size abstraction.

Rule §3.3 formulates the cost of an application of a function $f$ to argument $x$ given a size cost $n$, which we expect (and guarantee by the other rules) to be

$$\frac{\mathit{main}\ \mathrm{x}\ \mathrm{s}_1 \ \dot{\Longrightarrow}\ \mathrm{f}\ \mathrm{y}\ \mathrm{s}_2}{\mathit{TIME}\ \mathit{main}\ \_\ \_ \ \dot{\Longrightarrow}\ \mathit{TIME}\ \mathrm{f}\ (\mathit{SIZE}\ \mathrm{y}\ \{\}\ \{\})\ \mathrm{s}_2} \qquad \text{§3.1}$$

$$\frac{\mathrm{f}\ \mathrm{x}\ \mathrm{s}_1 \ \dot{\Longrightarrow}\ \mathrm{rhs}}{\mathit{TIME}\ (\mathrm{f}\ \mathrm{x}\ \mathrm{s}_1)\ \mathrm{n}\ \mathrm{s} \ \dot{\Longrightarrow}\ \mathit{TIME}\ \mathrm{rhs}\ \mathrm{n}\ \mathrm{s}} \quad \mathrm{f} \neq \mathit{main} \qquad \text{§3.2}$$

$$\frac{\mathit{TIME}\ (\mathrm{f}\ \mathrm{x}\ \mathrm{s}_1)\ \mathrm{n}\ \mathrm{s}_2}{\mathit{TIME}\ \mathrm{f}\ \mathrm{n}\ \mathrm{s}_1} \qquad \text{§3.3}$$

$$\frac{\mathit{TIME}\ (\_\ \_\mathrm{when}\_\ \mathrm{args})\ \mathrm{n}\ \mathrm{s}}{(\mathbf{foldl1}\ (+)\ \mathrm{times})\ \_\mathrm{when}\_\ \mathrm{sizes}} \qquad \text{§3.4}$$

where
  sizes = ...;
  times = ...;

$$\frac{\mathit{TIME}\ (\_\mathrm{case}\_\ (\_ \diamond \mathrm{fields})\ \mathrm{cs})\ \mathrm{n}\ \mathrm{s}_2}{\begin{array}{l}\mathit{TIME}\ \mathit{cacheIn}\ \mathrm{n}\ \{\ \mathit{ref} \Rightarrow \mathrm{fields}\ \} \\ +\ \_\mathrm{case}\_\ (\mathscr{R} \diamond \mathrm{fields})\ \ \mathrm{tcs}\end{array}} \qquad \text{§3.5}$$

where
  tcs = ...;

Figure 5: Program transformation rules mapping state semantics to time semantics.

the size abstraction of $x$. Note that while the various call semantics would require inclusion of different costs in this rule, we solved this problem by enforcing ANF: as function compositions are disallowed we now know that function arguments are evaluated (analyzed) at a separate program point. Without ANF this rule would have to be more complex. The size argument is only used in loop analysis: while do not perform loops analysis in this paper as it is currently not relevant for P4, we prepared the notation in preparation for future research.

Rule §3.4 transforms let expressions in the state semantics for let expressions in the time semantics. Let expressions assure size and time efficiency of cost formula evaluation as they can be used to eliminate redundant expressions and memoize intermediate results. While nested lambda expressions can be used for the same purpose, let expressions proved to be a far more human-readable alternative. ANF is a must in both cases. To aid readability, we omitted implementation details of the body of this rule, and instead recommend the reader to look at the input-output chart in Figure 6. The rule will bind the size abstraction of the program state after each operation in a sequence to variables in the let expressions. These variables are then utilized in the summation of the time costs of these operations that is returned by the expression. Rule §3.4 requires that bodies of let expressions in

the state semantics refer to a single bound variable of their parent let expressions (non-conforming let expressions can be easily translated to conforming ones). On Figure 6, we can observe the effect of the required changes: instead of featuring a sum of time costs in the body of the let expression, we only feature the size cost of the last application, which depends on the size cost of the expression before the last one, and so on.

Rule §3.5 rewrites a case analysis returning a state, to a case analysis returning a time expression. To aid readability, we omitted implementation details of the body of this rule, and instead recommend the reader to look at the input-output chart in Figure 6: Here, we first add the (target-dependent) potential cost ($c$) of caching the head value (i. e. reading from main memory to CPU cache), followed by conditionally adding the execution of executing the matching case body ($b_i$), plus the cost of the comparison between the head and the case pattern ($c_i$) (also taking into account the costs of all preceding matches). As the time semantics abstracted away program state information, we cannot precisely evaluate the case analysis anymore. Yet, we can still extract valuable information by applying statistical transformations, such as taking the maximum or the average of the case costs. We signified this by transforming the case head into a random variable ($\mathscr{R}$).

For brevity, we do not include the system transforming state semantics to size semantics, as it is mostly analogous to the system in Figure 5, but without the requirement to sum up the sizes of the sizes of intermediate operations.

```
TIME (x3
    _when_
      [ x1 --> f x0 sf
      , x2 --> g x1 sg
      , x3 --> h x2 sh
      ]) n s
```

(a)

```
(TIME f nx sf + TIME g ny sg + TIME h nz sh)
_when_
  [ nx --> SIZE x n  s
  , ny --> SIZE f nx sf
  , nz --> SIZE g ny sg
  ]
```

(b)

```
SIZE (x3
    _when_
      [ x1 --> f x0 sf
      , x2 --> g x1 sg
      , x3 --> h x2 sh
      ]) n s
```

(c)

```
SIZE h nz sh
_when_
  [ nx --> n
  , ny --> SIZE f nx sf
  , nz --> SIZE g ny sg
  ]
```

(d)

Figure 6: Example of time and size cost reductions of a let expression.

Finally in Figure 8, we present examples of two families of statistical rules. Such rules can be used to handle conditional control flows in partially reduced time cost expressions. For example, when applied to branching expressions §3.6 will return the cost of the most expensive branch (deriving the *worst case execution time*), while §3.7 will weigh the cost of each branch with the probability of the branch being executed times the probability that non of the preceding branches are being executed (deriving the *mean execution time*). All statistical rules behave as identity

```
TIME (_case_ (x ◇ fields)
        [ patt1 --→ b1
        , patt2 --→ b2
        , _     --→ b3
        ]) n s
```

(a)

```
c +
_case_ (ℛ ◇ fields)
  [ patt1 --→ c1 +          TIME b1 n s
  , patt2 --→ c1 + c2 +     TIME b2 n s
  , _     --→ c1 + c2 +     TIME b3 n s
  ]

where
 c  = TIME cacheIn n { ref ⇒ s ! fields };
 c1 = TIME cmp     n { ref   ⇒fields
                     , const ⇒ patt1 };
 c2 = TIME cmp     n { ref   ⇒fields
                     , const ⇒ patt2 };
```

(b)

Figure 7: Time cost reduction of a case analysis expression.

for non-branching expressions, as these are corresponding to one-element samples. Further statistics such as *best case execution time* and *variance* can be realized as similar rules.

$$\frac{MAX\ (\_\_\textbf{ifelse}\_\_\ \_\ t_1\ t_2)}{\textbf{max}\ t_1\ t_2}\ \S 3.6 \qquad \frac{AVG\ (\_\_\textbf{ifelse}\_\_\ c\ t_1\ t_2)}{(\mathscr{P}\ c)*t_1\ +\ (1-(\mathscr{P}\ c))*t_2}\ \S 3.7$$

Figure 8: Statistics rules for handling conditional control flow

# 4 Case study

In this section, we illustrate the system presented in the previous sections by going through the intermediate representations and target-dependent components used in the analysis of the parser in the small P4 program in Figure 1. To assist with this demonstration and keep this paper concise at the same time, we also provide a toy-sized target-dependent specification that can be substituted in the partial formula to obtain the final performance formula. In Figure 15, we illustrate a possible application of this performance formula to show how the estimated performance of this program on the specified platform changes w. r. t. the (application-dependent) probability of the transmitted packet being an IPv4 packet, and the (target-dependent) probability of cache misses.

## 4.1 State semantics transformation

In the first step of the analysis the P4 program is transformed to state semantics representation: a system of reduction rules describing the program as a composition of functions over the program state. We expect that production of such a representation is relatively straightforward after the P4 program was parsed into an IR, such as the one used by the P4C compiler [1]. Figure 9 depicts the reduction rules


Figure 9: Formal semantics of `start` and `parse_ipv4` generated from P4 program

corresponding to the `start` and `parse_ipv4` state transitions. Here, the input program state $x$ is modified by copying the bits at the current cursor position of `packet` to the memory segment specified by the `s!hdr!eth` field of the scope. Then, a value by the name in the `s!hdr!eth!ethType` variable is read from the resulting program state and is pattern matched to select the transition to the next parser state (formalized as a function application). The program state after executing the selected transition will be returned. We should note that the `extract` method in the `packet_in` extern is not defined in the P4 program (as it is target-dependent). If the state semantics is intended for execution, then an evaluation rule must be defined for this method. For cost analysis this is not required.

We may also note that transition `parse_eth` was eliminated in a compiler optimization step. As P4C was designed to separate target-independent P4 code optimizations from target-dependent ones into parts called *frontend* and *backend* respectively, we are free to rely on optimizations in the frontend, but are required to steer clear of those in the backends. This way, our cost estimations will always assume that any P4 compiler it is used with generates at least as efficient intermediate code as the P4C frontend. This assumption is automatically satisfied for P4 compilers realized as P4C backends, such as T4P4S [11].

In the next analysis step, the state semantics in Figure 9 is abstracted into time and size semantics using the program transformation system presented in Section 3. Figure 10 depicts the time semantics rule corresponding the `start` transition of the parser. We first calculate the $n_1$ size abstraction (a structure) of the program state after `extract` based on the size abstraction of the input program state $n$, and then the $n_2$ one after the case analysis (not depicted) based on $n_1$. Using these size abstractions, we can return the sum describing the execution costs of each operations and also the additional costs of the program control flow.

We defined the transformation rules so that the time cost semantics of a function call such as parameter passing (defined to be *copy-in/copy-out* by the $P4_{16}$ specification [15]) are included in the rule describing the function definition. This

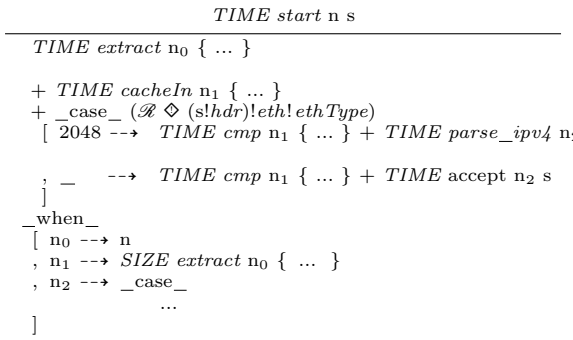



Figure 10: Time cost semantics of `start` derived using the system in Figure 5.

is the reason why the costs of the function do not appear in any of the applications (transitions do not require parameter passing). At this point, to represent the costs of the program control flow we also include the costs of reading an operand from memory into the cache, and the costs of performing the comparisons in the branches (the default case does not require a comparison, so in this case, only the preceding comparisons are counted).

## 4.2 Target dependent semantics

We utilize the semantic rules by applying them to concrete terms, i. e. function calls parameterized by a concrete program state. A model program state is depicted by Figure 11a, while the transformed size abstraction of this state is depicted by Figure 11b.

Conceptually, the concrete state is partially defined by the target, as it also describes the memory allocation scheme as prescribed by the target-specific compiler backend. For example, a backend implementing copy-in/copy-out semantics will allocate data-size memory for every function arguments, while another backend may choose to depart from the language specification and implement call-by-reference semantics with stacks to save both time and space. Moreover, parts of the concrete state may be explicitly target-dependent, such as the memory reserved for externs, and the formal representations of related storages (such as I/O buffers, L1, L2, L3 caches, NUMA memories). Figure 11a describes a program state in which the extern memory (called `packet_in`) reserved for storing a raw incoming packet is a 1 KB buffer and a pointer points to the position of the last parsed byte. As extern memory is ultimately target-dependent, we modeled this structure after the identically named C structure in Figure 12a.

It also includes the `header` structure declared in the original P4 program code with fields having the respective sizes. For simplicity, we omitted more intricate details, such as copy-in/copy-out semantics for this model. The state also includes

```
{ headers ⇒
    { eth  ⇒
        { ethType ⇒ mkarray 0 2
        , dstAddr ⇒ mkarray 0 6
        , srcAddr ⇒ mkarray 0 6
        }

    , ipv4 ⇒
        {
          ...
        }

    }
, packet_in ⇒
     { cursor ⇒ 0
     , buffer ⇒ mkarray 0 1000
     }
, cacheLineSize ⇒ 64
, cpuWordLength ⇒ 8
, cache          ⇒ mkarray 0 32000
, mem            ⇒ ...
, nic            ⇒ ...
, ...
}
```

(a) A simplified model of a concrete P4 program memory state. Sizes are given in bytes.

```
{ headers ⇒
    { eth  ⇒
        { ethType ⇒ 2
        , dstAddr ⇒ 6
        , srcAddr ⇒ 6
        , sizeof  ⇒ 14
        }
    , ipv4 ⇒
        { ...
          sizeof ⇒ 20
        }
    , sizeof ⇒ 34
    }
, packet_in ⇒
     { ...
       sizeof ⇒ 1000
     }
, cacheLineSize ⇒ 64
, cpuWordLength ⇒ 8
, cache          ⇒ 32000

, ...
, sizeof = ...
}
```

(b) Size abstraction of the P4 program state in Figure 11a.

Figure 11

explicitly target-dependent segments such as the 32 KB sized L1 `cache` field. Since we are working with small packets, we may assume infinite RAM memory without losing practical soundness.

Unless we want to execute the state semantics, we do not need the concrete program state. We described it to make it easier to understand its size abstraction. Figure 11b depicts this size abstraction. Abstracting the concrete state is usually non-trivial: P4 structures get a special field (denoted here as `sizeof`) storing its aggregated sizes, target-dependent constants are kept as is, and – to enable loop analysis in the packet parser – the size abstraction of `packet_in` is the size of the yet unprocessed part of the packet. As we do not yet perform loop analysis that would require intricate size abstractions, we left answering the questions of state abstraction for future research.

Any cost semantics derived from P4 code only cannot be a complete description of program behavior: as unspecified P4 constructs are defined by targets, we require target-specific information about how this target implements the unspecified P4 constructs (such as the pipeline and externs, as seen before). Figure 12b provides a partial example that formally specifies such target-specific information.

Note that we devised the target model in this section manually: while we suspect it to be reasonable, it is far too simplistic to be used for predicting real targets with common but advanced low-level features (such as NUMA, DMA, multiple cores, etc.).

```
typedef unsigned char byte;

typedef struct packet_in {
  byte buffer[1024];
  byte* cursor;
} packet_in;



void extract(packet_in* packet,
             void* hdr,
             unsigned long hdrLen) {

  memcpy(hdr, packet->cursor, hdrLen);

  packet->cursor = packet->cursor + hdrLen;

}
```

(a) An implementation of the P4 extern method `extract` in C.

$$
\begin{array}{l}
\textit{TIME extract}\ \mathrm{n}\ \mathrm{s} \\
\hline
\textit{TIME cacheIn} \\
\quad \mathrm{n} \\
\quad \{\ \textit{ref} \Rightarrow \mathrm{s}!\textit{packet}\ \} \\
+\ \textit{TIME memcpy} \\
\quad \mathrm{n} \\
\quad \{\ \mathrm{src} \Rightarrow \mathrm{s}!\textit{packet} \\
\quad ,\ \textit{dst} \Rightarrow \mathrm{s}!\textit{hdr}\ \} \\
+\ \textit{CPU\_ADD}
\end{array}
\qquad \S 4.2.1
$$

$$
\begin{array}{l}
\textit{TIME memcpy}\ \mathrm{n}\ \mathrm{s} \\
\hline
\textbf{ceil}\ (\mathrm{d}/\mathrm{l}) \\
* \\
(\textit{L1\_TO\_CPU} \\
+\ \textit{CPU\_MOV} \\
+\ \textit{L1\_FROM\_CPU}) \\
\\
\text{where} \\
\quad \mathrm{d} = \mathrm{n}\diamondsuit(\mathrm{s}!\textit{dst}); \\
\quad \mathrm{l} = \mathrm{n}\diamondsuit\textit{cpuWordLength};
\end{array}
\qquad \S 4.2.2
$$

(b) Cost model based on 12a.

Figure 12

Target-dependent time semantics may be delivered automatically from a sufficiently formal target specification, but – due to the lack of various invariants guaranteed by P4, such as compile-time known memory requirements and no loops or upper-bounded loops only – we deem this problem to be non-trivial and out of the scope of this paper.

In Rule §4.2.1 of Figure 12b, we model the unspecified `extract` operation that attempts to parse a packet header (i. e. copies bits of the incoming packet to a memory segment representing a header), as a call to a system-level copy operation such as C's `memcpy`, followed by incrementing the counter by the size of the parsed data (see Figure 12a). As a simplification, we assume that only valid (as in, parseable) packets arrive: if this were not the case (as usually), the rule would have to be extended with the cost of checking for input end and also the early return should be calculated in.

Our cost model of `memcpy` in Rule §4.2.2 assumes that the part of packet under operation is already cached (and fits entirely in the cache, which is reasonable for $32k$ cache size), and then read its 64 bit size chunks (size of the CPU registers) into the CPU registers with the aim of performing the CPU-level copy instruction. The size of part is the compile-time known destination size, i. e. the size the header we want to parse from the raw packet. Note that we may model the cost of comparison (used in calculating the cost of case analysis) very similarly, with the only difference that we need to read two words into the CPU registers instead of one, and perform

the CPU-level comparison operation instead of copying.

| **Description** | **LHS** | **RHS (cycles)** |
|---|---|---|
| Cost of comparing the contents of two CPU registers. | *CPU_CMP* | 1 |
| Cost of copying data between CPU registers. | *CPU_MOV* | 2 |
| Cost of addition with a constant number. | *CPU_ADD* | 5 |
| Cost of copying a word from L1 to a CPU register. | *L1_TO_CPU* | 5 |
| Cost of copying a word from a CPU register back to L1. | *L1_FROM_CPU* | 5 |
| Cost of copying a cache line from memory to L1 cache. | *MEM_TO_CACHE* | 79 + 200 |

Figure 13: CPU architecture model[1] specifying CPU instruction execution costs.

In Figure 13, we defined all cost constants required for completely reducing the preceding formulae, based on external specifications and benchmarks of the respective operations of the selected CPU architecture.

As expected, the costs are seemingly dominated by the reads from memory to caches, but we should note that this operation handles cache lines (64 byte in in our example), while the others handle register-sized data (64 bit in our example) and thus will be repeated in succession for larger data (so if copying 8 byte from cache to CPU registers requires 5 cycles, the same operation for 64 byte will require 40 cycles in this model).

## 4.3 Symbolic time cost formula and applications

At this point, we introduced most of the basic components required for evaluating the time cost of `start`, given the abstracted state in Figure 11b. We merge the cost semantics generated from P4 with the target-specific cost semantics to derive an intermediate formula (not depicted here because of its size) from the cost expression of `start`, and finally apply our chosen statistics transformation.

Figure 14 depicts the worst case execution time of `start` we derived using Rule §3.6. This approximates the execution time of `start` when every incoming packet is an IPv4 packet (i. e. `ethType` is `2048`) and the cache misses in each attempt.

The best case execution time of `start` is the formula: 8 * (`L1_TO_CPU` + `CPU_MOV` + `L1_FROM_CPU`) + `CPU_ADD` + 1 * (2* `L1_TO_CPU` + `CPU_CMP` + `L1_FROM_CPU`). This is the execution time when no incoming packets are IPv4 packets, and the cache hits in each attempt.

We can derive the average case similarly, but instead of taking each member for granted, we appropriately have to weigh costs corresponding to parts of the code with with the probabilities of that part being executed. For example, the costs of

[1]Based on Intel Skylake X (4.3 GHz, 32KB L1) [4, 2]. *MEM_TO_CACHE* value was adapted assuming 0.25ns per cycle.

| | |
|---|---|
| *TIME extract* $n_0$ { ... } | |
| • Cost of reading the packet into cache | *MEM_TO_CACHE* |
| • Cost of extracting the header | + 8 ∗ (*L1_TO_CPU*<br>+ *CPU_MOV*<br>+ *L1_FROM_CPU*) |
| • Cost of incrementing the cursor | + *CPU_ADD* |
| Case analysis (WCET) | |
| • Cost of reading the case head into cache | + *MEM_TO_CACHE* |
| • Cost of comparing the case head with the pattern of the most expensive case (i. e. the first one). | + 1 ∗ (2∗*L1_TO_CPU*<br>+ *CPU_CMP*<br>+ *L1_FROM_CPU*) |
| *TIME parse_ipv4* $n_2$ s | + *MEM_TO_CACHE*<br>+ 8 ∗ (*L1_TO_CPU*<br>+ *CPU_MOV*<br>+ *L1_FROM_CPU*)<br>+ *CPU_ADD* |

Figure 14: The WCET of state `start`.

the `parse_ipv4` state transition is weighed with the probability of the `ethType` field of the header being the value `2048`. For conciseness, we omit the resulting expected value formula from this paper, and instead show only the final values in Figure 15, given various probability distributions.

Using the constants in Figure 13, we can finally evaluate the partially evaluated formulas to a single numeric value characterizing the performance. In Figure 15, we depicted three constants and the average cost computed over various probabilities. The constant values marked with `BCET` and `WCET` denote the best and worst case execution times of `start` as we discussed earlier. `TOL` was introduced in Section 1.1 as the maximum number of cycles that can be spent for processing a packet without causing buffer overflow in the long run by a switch residing in a 10 Gigabit Ethernet network and implemented on the CPU architecture in Figure 13. Note that `TOL` encompasses the entirety of the packet processing process starting with packet arrival on the NIC, while the rest of the numbers only characterize the costs of packet parsing starting from `start`. To derive a more meaningful chart, we would need to reduce the time cost formula for the program entry point instead of `start`, or set `TOL` lower.

Instead of deriving the average execution time with arbitrary probability distributions, we plotted the average (measured in cycles) over several different distributions. Because of the case analysis and the possible need of caching, the average depends on the probability of the packet being an IPv4 packet, and the probability of cache misses. This means that we are working with pairs of probability distributions, each defined over two values (the predicate in question being true and false). Two keep the plot in 2 dimensions, we only used two probabilities of packets having IPv4 types (0 meaning no IPv4 packets arrive at all, and 1 meaning only IPv4 packets arrive), each represented by two different lines. We keep track of the cache miss probabilities on the $x$-axis.

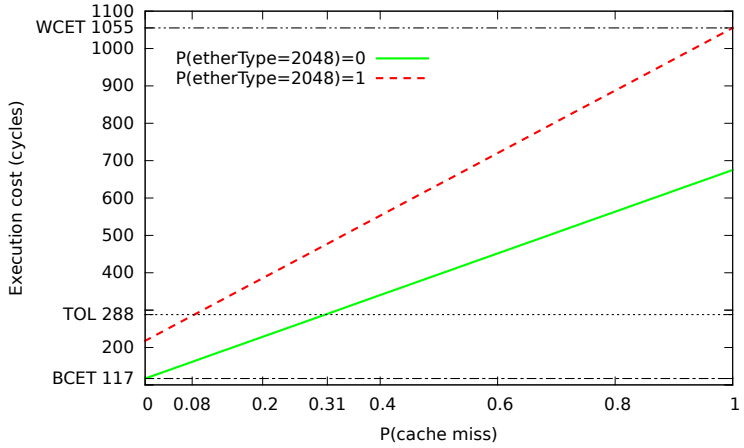



Figure 15: Latency characteristics of `start` over IPv4 and cache miss probabilities.

By looking at the plot, we can conclude that a cache miss ratio of 0.08 and below will guarantee that the examined P4 function call will not take longer than the allowable limit even if we have to process all packets as IPv4 packets. On the other hand, ratios above 0.31 are clearly dangerous: if cache the misses that frequently, buffer overflows cannot be avoided even if no IPv4 packets arrive at all. As these extremities rarely occur in practice, cache miss ratios between 0.08 and 0.31 can be candidates for further testing (or cost analyses with more precise estimations) to find the highest cache miss ratio with which the switch can still run sustainably (i. e. without causing buffer overflows) in a given application environment.

# 5 Related work

In this paper, we presented a system for verifying conformance of $P4_{16}$ programs to performance requirements. Our long term goal is to innovate a framework that is capable of verifying both functional and non-functional requirements using the same base representation. To handle cost analysis methodically, we utilized the seminal work of Wegbreit [16] and the idea of cost relation systems (CRS) [5].

Verification of functional correctness of P4 programs seems to be hot topic lately in the switching industry and the field of network languages, although most works we found were targeting $P4_{14}$ the previous, still maintained version of the language. *P4V* [12] verifies various properties – such that no headers are used unless they were extracted from a packet beforehand – by extending the language with assertion statements, transforming the program code including assertions to predicate transformer semantics, and then applying the Z3 SMT solver to prove theorems. *Vera* [14] follows a different route, and uses symbolic execution and

computation tree logic over an intermediate representation to find or prove non-existence of bugs in P4 programs, delivering also an input configuration producing the fault. *P4K* [10] is a formal semantics for P4, written in the K framework. Using reachability logic in K, the authors automatically prove Hoare-style assertions for P4 involving stateful data plane elements and unbounded streams of packets.

Both in correctness and performance verification a core concern is efficiency: for deriving the execution cost of a program statically, analysis of all execution paths is unavoidable. This means that the complexity of the analysis is at best exponential. To offset the costs of path analysis, we borrowed a simple divide and conquer idea from [8] to utilize the highly decomposable nature of network programming languages in verification: instead of analysing paths in the full program, we first analyze just the components, and only perform those transformations on the full program that actually require the full program. Thanks to our symbolic and compositional denotation, we can choose an arbitrary small segments for analysis instead of analyzing the full pipeline.

Our work can be considered an automatization of the approach following [3]. Here, the authors manually analyze the Ethernet protocol and a specific hardware architecture, then synthesize the information into a sequence of primitive packet processing actions called *elementary operations (EOs)* in order to quantify performance.

## 6 Conclusion and Future Work

We conclude this paper by first enumerating problems and opportunities to extend the presented framework in future research, and then summarizing the contributions of this work.

In the current paper, we only analyzed the parser of a P4 program. One important step for full language coverage will be the analysis of match-action tables: match probabilities can be computed from given match-action tables, and low-level costs of matching and actions can be inferred using the presented procedure. On the other hand, the number of branches in the control flow will be equal to the number of distinct actions that can be performed by the table, so to avoid combinatorial explosion with nested branches, it is important to analyse match-action tables separately.

In this paper, we demonstrated the operation of our system on a toy example. It will be an important and useful research task to apply the procedure to real and complex targets, such as the P4C reference switch [1] and the DPDK switch generated by T4P4S [11]. First, to validate the presented system in real environment, and second to use the retrieved performance information to improve the aforementioned targets.

By introducing random variables in time cost formulas, we effectively modelled P4 programs as memoryless Markov-chains. Feasibility of providing conditional probability distributions for more precise models involving Markov-chains with longer memory may also worth further investigation.

At the time of writing, state-of-art compilers such as the official P4C compiler [1] as well as the P4C-based T4P4S reject all P4 programs containing loops in the parser, and the language specification [15] disallows loops everywhere else. For the lack of support in the software ecosphere and a seeming lack of use cases for loops in P4, we decided not to implement cost analysis of loops in the current work, but we intentionally choose a representation that can be extended for this purpose applying approaches involving e. g. generating functions [16], or cost relation systems [5], and also see loop analysis a possible direction towards system completeness.

With this, we conclude our paper. We showed that networked switches have to comply with strict performance requirements, and also observed that unspecified constructs in P4 require low-level, target-dependent information. We outlined the architecture of a cost analysis framework for addressing both problems. We presented a program transformation system based on term rewriting, that is used to derive a symbolic formula, in which the symbols can be substituted in with numeric constants by various queries to deliver the requested performance information. We went through the main steps of this process using a toy example, and showcased a possible application of the symbolic formula to find ideal cache miss ratios for the aforementioned target. We also situated our paper among works related to the verification of P4, and network function cost analysis. Finally, we marked possible directions to improve this work in order to provide a practical solution for analysis and verification of high-efficiency network platfroms.

## References

[1] P4C reference compiler for the $P4_{16}$ programming language. https://github.com/p4lang/p4c, 2017. [Online; accessed 30-September-2018].

[2] 7-Zip LZMA Benchmarks. https://www.7-cpu.com/cpu/Skylake_X.html, 2018. [Online; accessed 30-September-2018].

[3] A. Sapio and M. Baldi and G. Pongrácz. Cross-Platform Estimation of Network Function Performance. In *2015 Fourth European Workshop on Software Defined Networks*, pages 73–78, Sept 2015. DOI: 10.1109/EWSDN.2015.64.

[4] Agner Fog. Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD and VIA CPUs. https://www.agner.org/optimize/instruction_tables.pdf, 2018.09.15. [Online; accessed 30-September-2018].

[5] Albert, Elvira, Arenas, Puri, Genaim, Samir, and Puebla, Germán. Cost relation systems: A language-independent target language for cost analysis. *Electron. Notes Theor. Comput. Sci.*, 248:31–46, August 2009. DOI: 10.1016/j.entcs.2009.07.057.

[6] Albert Gräf. The Pure Programming Language and Library Documentation. https://agraef.github.io/pure-docs/, 2018. [Online; accessed 30-September-2018].

[7] Bosshart, et. al. P4: Programming protocol-independent packet processors. *SIGCOMM Comput. Commun. Rev.*, 44(3):87–95, July 2014. DOI: 10.1145/2656877.2656890.

[8] Dobrescu, Mihai and Argyraki, Katerina. Software dataplane verification. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, NSDI'14, pages 101–114, Berkeley, CA, USA, 2014. USENIX Association.

[9] Flanagan, Cormac, Sabry, Amr, Duba, Bruce F., and Felleisen, Matthias. The essence of compiling with continuations. In *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation*, PLDI '93, pages 237–247, New York, NY, USA, 1993. ACM. DOI: 10.1145/155090.155113.

[10] Kheradmand, Ali and Rosu, Grigore. P4K: A formal semantics of P4 and applications. *CoRR*, abs/1804.01468, 2018.

[11] Laki, Sándor, Horpácsi, Dániel, Vörös, Péter, Kitlei, Róbert, Leskó, Dániel, and Tejfel, Máté. High speed packet forwarding compiled from protocol independent data plane specifications. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM '16, pages 629–630, New York, NY, USA, 2016. ACM. DOI: 10.1145/2934872.2959080.

[12] Liu, et al. P4V: Practical Verification for Programmable Data Planes. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '18, pages 490–503, New York, NY, USA, 2018. ACM. DOI: 10.1145/3230543.3230582.

[13] Sivaraman, Anirudh, Kim, Changhoon, Krishnamoorthy, Ramkumar, Dixit, Advait, and Budiu, Mihai. Dc.p4: Programming the forwarding plane of a data-center switch. In *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research*, SOSR '15, pages 2:1–2:8, New York, NY, USA, 2015. ACM. DOI: 10.1145/2774993.2775007.

[14] Stoenescu, et. al. Debugging p4 programs with vera. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '18, pages 518–532, New York, NY, USA, 2018. ACM. DOI: 10.1145/3230543.3230548.

[15] The P4 Language Consortium. $P4_{16}$ Language Specification. https://p4.org/specs/, 2017. [Online; accessed 30-September-2018].

[16] Wegbreit, Ben. Mechanical program analysis. *Commun. ACM*, 18(9):528–539, September 1975. DOI: 10.1145/361002.361016.

Acta Cybernetica **24** (2019) 83–103.

# Multi-Cloud Management Strategies for Simulating IoT Applications

Andras Markus[a] and Jozsef Daniel Dombi[b]

**Abstract**

The Internet of Things (IoT) paradigm is closely coupled with cloud technologies, and the support for managing sensor data is one of the primary concerns of Cloud Computing. IoT-Cloud systems are widely used to manage sensors and different smart devices connected to the cloud, hence a large amount of data is generated by these things that need to be efficiently stored and processed. Simulation platforms have the advantage of enabling the investigation of complex systems without the need of purchasing and installing physical resources. In our previous work, we chose the DISSECT-CF simulator to model IoT-Cloud systems, and we also introduced provider pricing models to enable cost-aware policies for experimentation. The aim of this paper is to further extend the simulation capabilities of this tool by enabling multi-cloud resource management. In this paper we introduce four cloud selection strategies aimed to reduce application execution time and utilization costs. We detail our proposed method towards multi-cloud extension, and evaluate the defined strategies through scenarios of a meteorological application.

**Keywords:** cloud computing, internet of things, simulation, Pliant system

## 1 Introduction

In the paradigm of the Internet of Things (IoT), sensors and smart devices are connected to the Internet giving way to many opportunities to use cloud and IoT services together [1]. Since more and more devices enter the network to form IoT systems, the dataflow and the workload of the supporting services are increasing, which also raise open issues such as resource usage and cost reduction or legal compliance [5]. Hiring physical machines from virtual server parks fitting various IoT scenarios could be very expensive, and the investigation of IoT-enabled cloud service compositions is not always possible with real cloud providers. As a result, in

[a]Software Engineering Department, University of Szeged, 6720 Szeged, Hungary, E-mail: markusa@inf.u-szeged.hu

[b]Software Engineering Department, University of Szeged, 6720 Szeged, Hungary, E-mail: dombijd@inf.u-szeged.hu

DOI: 10.14232/actacyb.24.1.2019.7

many cases cloud simulators are applied to address the evaluation of such complex environments.

While network simulators could be too complex to simulate IoT and cloud systems together, due to detailed network configurations, special purpose cloud simulators may be over-tailored to cloud-specific details making it hard to express IoT needs. The number of IoT devices and usage areas are constantly growing, and some cases require immediate intervention after data processing, such as heart monitoring in smart homes, or traffic control in smart cities. This means we need new solutions and techniques for data storage, access and processing, which can be designed and evaluated in infrastructure cloud simulators extended with IoT simulation capabilities. Therefore we have chosen DISSECT-CF to perform our investigations [2].

In our earlier works we introduced IoT modeling to a traditionally cloud simulator, then combined provider pricing schemes with IoT cloud management in DISSECT-CF [3] to enable cost-aware investigations. Since cloud federations [4] provide a wider range of capabilities to users, the next step in our research was to enable the usage of multiple cloud datacenters to serve certain IoT scenarios.

In this paper we introduce four cloud selection strategies aimed to reduce application execution time and utilization costs. The default strategy uses random cloud selection for the managed IoT devices, and we also propose a load balancing and a cost minimizing strategy. Finally, for a more sophisticated strategy we apply a fuzzy-based approach. We also evaluate our proposal through scenarios derived from a real-life weather forecasting service.

The remainder of this paper is as follows: Section 2 discusses related approaches in this field, and Section 3 summarizes relevant previous works of the authors. Section 4 introduces our proposed cloud selection strategies, which are evaluated in Section 5. Finally, we conclude the paper in Section 6.

## 2 Related work

In the field of cloud and IoT simulations, CloudSim [6] is one of the most widespread solutions for modeling cloud system components including data centers and virtual machines, as well as investigating cloud resource provisioning policies. When IoT started to emerge, CloudSim has been extended to provide modeling capabilities for IoT system components. Khan et al. [7] proposed an infrastructure coordination technique for large scale IoT systems built on top of CloudSim. It provides customization for specific home automation scenarios, which limits the applicability of their extensions. The iFogSim [8] also extends CloudSim to simulate IoT and fog environments by measuring resource management techniques with several metrics including latency, network congestion, energy consumption and cost. They presented two case studies to demonstrate IoT modeling and resource management policies: latency-sensitive online gaming, and an intelligent surveillance application using distributed camera networks.

Besides CloudSim, we can find similar simulation approaches tailored to specific

needs. Zeng et al. [13] proposed IOTSim that supports the simulation of big data processing with the MapReduce model exemplified with a real case study. SimIoT [9] is based on the SimIC simulation framework [10], and it proposes several techniques to simulate the communication possibilities between IoT sensors and cloud components, but it is limited to compute activity modeling.

MobIoTSim [14] proposes a semi-simulated environment for investigating IoT cloud systems. It aims at mimicking the behavior of IoT sensors and devices with a mobile simulation environment. Sensor data management and system scalability can be investigated with real interconnected gateway services.

Concerning IoT management algorithms, Moschakis and Karatza [11] introduced workload models with interfacing various cloud providers and IoT systems, enabling the investigation of the behavior of cloud systems that support the processing of data originated from the IoT system. Silva et al. [12] focused on the dynamic nature of IoT systems, therefore they investigated fault behaviors with specific fault models. Unfortunately, the scalability of the introduced fault behavior concepts are insufficient for large scale systems.

Several providers offer PaaS-level cloud services with the possibility of connecting and managing IoT devices, we can find a detailed comparison of them in [15]. These solutions are usually tightly coupled for certain providers, and hide low-level details of utilization, which is an advantage for end-users, but they are not suitable for modeling low-level infrastructure operations, and developing multi-provider IoT-Cloud applications.

From these related works we can see that IoT-Cloud systems can be examined with several simulation tools, and supporting environments already exist for investigating specific behavioral methods, such as resource selection, sensor communication, big data management, energy efficiency and cost savings. Nevertheless, the combination of these aims and closer relation to real world utilization patterns still represent open issues. Our approach combines cost reduction based on real world provider pricing and multi-cloud resource selection, applied in a real-world usage scenario.

# 3 IoT-Cloud Simulation in DISSECT-CF

One of our main goals for choosing the DISSECT-CF cloud simulator for our investigations was its unified resource sharing mechanism. Timed events are the basic elements of this simulator, which can be recurrent time-dependent events that have a frequency value (e.g. 10 ms), which calls their methods regularly in every moment based on the given value. There are non-recurrent time-dependent events as well, that have only a delay value (e.g. 5 ms) denoting the time to be elapsed before its function has to be called. Both type of events are controlled by the inside clock of the simulator. With these build-in events we can simulate the management of IoT systems including sensors and smart devices. The configuration of IoT system properties in the simulator can be done through an XML description file. We can set the following attributes: network bandwidth, local repository size, operating time,

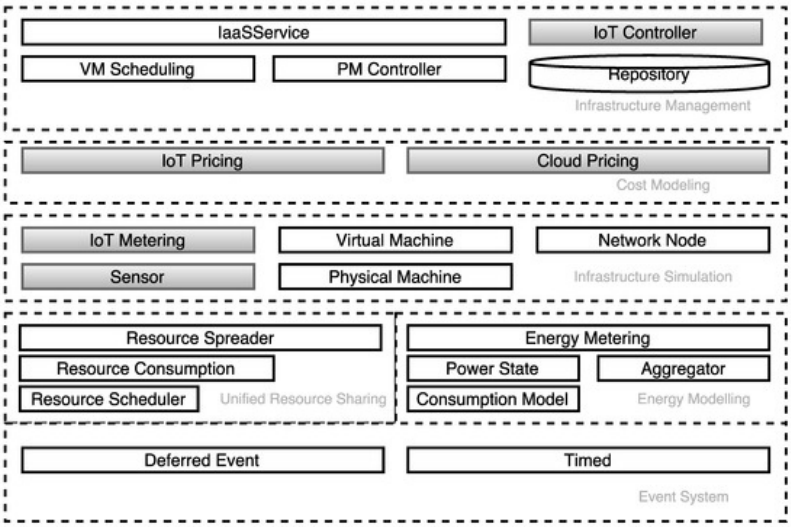



Figure 1: Architecture of the IoT-extended DISSECT-CF simulator

number of sensors and frequencies of the data generating, storing and sending, and the size of the generated files.

Two additional XML descriptions can be used to set provider pricing properties. Usually the cloud side pricing is used to calculate the costs of virtual machines (VMs) used to run an IoT application. It defines a fixed monthly cost per VM instance, but some providers charge the hour per price for every instance an application needs. To manage data coming from IoT devices and sensors, we need to calculate the IoT side costs, that can also be set based on real provider pricing schemes (e.g. Amazon, Azure, IBM and Oracle). In general, the IoT prices are calculated after the generated data traffic in MB following the "pay as you go" approach, while some providers charge after the number of messages exchanged in a month, or set a monthly device per price or messages sent in a day. All these three XML description formats and possible parameters are presented and discussed in our previous work [3].

In general, a simulation is performed by executing the following steps: first, a cloud is set up using an XML description (we used the model of a Hungarian private cloud infrastructure called the LPDS Cloud of MTA SZTAKI [24]), then the necessary amount of stations are initialized and the VM parameters are loaded from additional XML files, which also describe the cloud and IoT costs. Next, the IoT application is started with the deployment of an initial VM in the defined cloud, followed by the start of metering and data generation processes of device stations. IoT and cloud operations are continuously monitored to calculate the resource consumption costs. During execution, a broker service checks if the cloud repository received a scenario-specific amount of data, if so, then a compute task

will be generated and deployed in a VM for data processing.

Finally, sensor data generation, compute task creation and execution are repeated till the end of the simulation, with possible starting and stopping of VMs. At the end of the simulation we can retrieve information about end user costs concerning the utilization of IoT and cloud resource consumption.

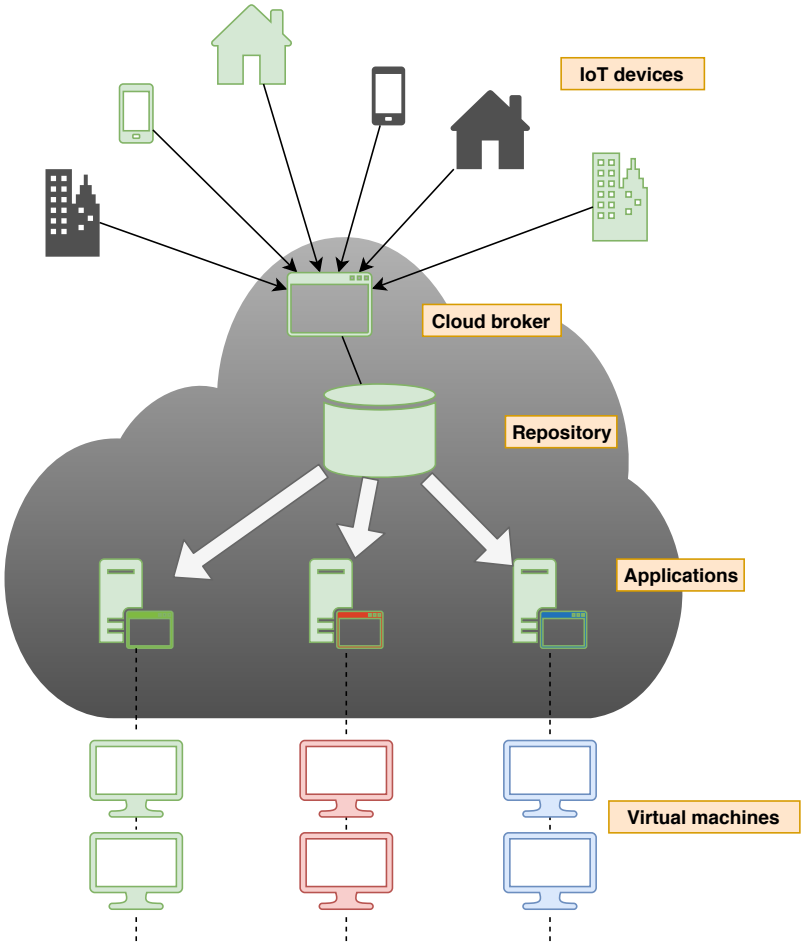



Figure 2: Application execution in the extended DISSECT-CF simulator

## 4 The proposed cloud selection strategies

The main research question of this paper is how we can influence the behavior of an IoT application, if the sensors can have different allocation strategies for multiple clouds. In the earlier version of the extended DISSECT-CF we could exploit only

one cloud datacenter to start VMs, therefore all sensors and smart devices were connected to this specific cloud, and all the generated data of the sensors were processed by virtual machines running in the same cloud (as we summarized in the previous section). In this single cloud setup, a cloud can have a preloaded cost calculation policy with a single pricing scheme. Smart devices usually have different sensors and usage frequencies affecting data generation methods that can influence cloud service operation and also the provider pricing. As a result, a single cloud could be easily overloaded, and the unprocessed data could hinder the operation of the IoT application causing longer response times, even service unavailability in real-world services.

The formerly added components of the simulator by our previous work towards IoT extension can be seen in Figure 1 denoted by grey background. Our current contribution targets the top levels of the simulator architecture, and aims to enable the use of multiple IoT Controllers and pricing schemes.

In this work we introduce the possibility of multi-cloud management for IoT cloud simulations in DISSECT-CF. During the start of the simulation we can set up different clouds using extended XML descriptions denoting sets of physical machines and repositories with various properties. Another improvement is the introduction of a cloud broker, which can manage different VM queues. These queues may have virtual machines with different pricing policies, and within a simulation the broker can decide, to which cloud (and to which VM queue) the IoT devices should be connected, thus where the generated data should be sent and processed in an application. This revised IoT Cloud management architecture is depicted in Figure 2, showing one cloud with three different applications mapped to three different VM queues. These extensions make the simulator more flexible and capable of performing scalability experiments involving multiple cloud providers.

In order to enable easy and repeatable system configuration, we defined a new XML format to configure VM flavors with prices for the applications. An example of this XML structure can be seen in Figure 3, which defines two flavors. With this flavor model we can specify the required VM resources (*cpu-cores, ram*), the cost of the VM (*price-per-tick*), the number of boot instructions affecting the boot time (*startup-process*), the network traffic (*network-load*) and the local disk size requirement (*req-disk*) of the new instance. Flavors can be identified by the *name* attribute, and they must be unique in the configuration. In this example we defined two different clouds (4 CPU cores with 4 GB RAM, and 2 CPU cores with 2 GB RAM), which allocate almost 10 GBs of the local disc.

An example XML with two application descriptions is presented in Figure 4. We defined a daemon service frequency (*freq*) to regularly check the repository for unprocessed data. The *tasksize* attribute tells the highest amount of unprocessed data that can packaged in one compute task to be executed by virtual machines. The selected computing infrastructure is identified by the *cloud* tag, and finally, the VM flavor to be used for executing the compute tasks can be specified in the *instance* tag (by referring to the *name* attribute of the flavor model).

In this example both defined applications use the formerly defined flavors and two different clouds (identified by 'Cloud1' and 'Cloud2' unique name). The fre-

```
<?xml version="1.0"?>
<flavors>
   <flavor name="amazon-large">
     <ram>4294967296</ram>
     <cpu-cores>4</cpu-cores>
     <price-per-tick>0.000015</price-per-tick>
     <core-processing-power>0.001</core-processing-power>
     <startup-process>100</startup-process>
     <network-load>0</network-load>
     <req-disk>10000000000</req-disk>
   </flavor>
   <flavor name="azure-small">
     <ram>2147483648</ram>
     <cpu-cores>2</cpu-cores>
     <price-per-tick>0.000001</price-per-tick>
     <core-processing-power>0.001</core-processing-power>
     <startup-process>100</startup-process>
     <network-load>0</network-load>
     <req-disk>10000000000</req-disk>
   </flavor>
</flavors>
```

Figure 3: Sample description using the flavor XML model

quency value defines that the daemon service should repeat the virtual machine handling functions (generate, shutdown and reboot the VM based on the actual load of unprocessed data) every 5 minutes.

In the IoT paradigm the sensors are passive entities of the systems, thus their performance is limited by the operation frequency (i.e., data generation, storing, transfer to the cloud), up-time and network connection. Usually, large amounts of sensor data are sent from the smart devices to cloud resources for further computation and analysis. Since resource consumption can be costly, IoT application owners can reduce their expenses by selecting a provider having a suitable pricing scheme.

In this paper we defined four different strategies to perform cloud provider selection (to be done during each IoT device (or sensor) start-up), which can be denoted by setting the strategy field of the XML description of each device participating in the simulation. The strategy for a smart device is defined in the device XML description format presented in Figure 5 with the *strategy* tag. Also the network settings (*maxinbw, maxoutbw, diskbw*) of the local repository (*reposize*) are set with data caching function (*data-ratio*). We can configure the life time of

```
<?xml version="1.0"?>
<applications>
   <application tasksize="2500000">
      <name>Weather-1</name>
      <freq>300000</freq>
      <cloud>cloud1</cloud>
      <instance>azure-small</instance>
   </application>
   <application tasksize="2500000">
      <name>Weather-2</name>
      <freq>400000</freq>
      <cloud>cloud2</cloud>
      <instance>amazon-large</instance>
   </application>
</applications>
```

Figure 4: Sample description using the application XML model

the device (*starttime, stoptime*), the number of sensors it has (*sensor*), the size of the generated data (*filesize*) and the generation and sending frequency (*freq*). The device settings can be applied for a group of devices using the *number* attribute. In this example (Figure 5) the configuration file defines 487 devices running for 6 hours and each device generates 50 bytes of data by its 8 sensors. Detailed XML samples and schemes can be found in [20].

We propose four different strategies for multi-cloud management: (i) *random*, (ii) *cost-aware*, (iii) *runtime-aware* and (iv) *Pliant*. In the next subsections we introduce these strategies.

## 4.1 Basic strategies

With the *random* strategy the cloud broker chooses one of the available applications running in the simulated clouds randomly for an actual IoT device (sensor or station).

The *cost-aware* strategy looks for the cheapest available VM in a cloud (based on their static pricing properties), thus it compares the prices of the required VM flavors for a given device. Its algorithm orders the VMs by their price-per-tick value. This solution may be more suitable for IoT applications having relatively small data processing needs or less susceptible for the processing time, because cloud providers usually offer lower resource capacities for less costs.

In the *runtime-aware* strategy, the corresponding algorithm ranks the available VMs (residing in different clouds) by a specific value defined by the ratio of the

```
<?xml version="1.0" encoding="UTF-8"?>
<devices>
   <device starttime="0" stoptime="21600000"
           number="487" filesize="50">
      <name>test1</name>
      <freq>60000</freq>
      <sensor>8</sensor>
      <maxinbw>1000</maxinbw>
      <maxoutbw>1000</maxoutbw>
      <diskbw>1000</diskbw>
      <reposize>60000</reposize>
      <data-ratio>1</data-ratio>
      <strategy>random</strategy>
   </device>
</devices>
```

Figure 5: Sample description using the device XML model

number of already connected devices and the number of the available physical machines of the hosting cloud. This is a dynamic strategy taking into account the actual load of the available clouds. Applications having longer data processing needs may prefer this strategy.

## 4.2 The Pliant strategy

Fuzzy sets were introduced in 1965 with the aim of reconciling mathematical modeling and human knowledge in the engineering sciences. Fuzzy logic means that we can not decide whether the value is true or not. The true lies between the true and false value. Fuzzy logic offers a very valuable flexibility for reasoning [17]. Most of the building blocks of the theory of fuzzy sets were proposed by Zadeh, especially fuzzy extensions of classical basic mathematical notions like logical connectives, rules, relations and quantifiers. Over the last century, fuzzy sets and fuzzy logic [16] have become more popular areas for research, and they are being applied in fields such as computer science, mathematics and engineering. This has led to a truly enormous literature, where there are presently over thirty thousand published papers dealing with fuzzy logic, and several hundreds books have appeared on the various facets of the theory and the methodology. However, there is not a single, superior fuzzy logic or fuzzy reasoning method available, although there are numerous competing theories.

The Pliant system is a kind of fuzzy theory that is similar to a fuzzy system [18]. The difference between the two systems lies in the choice of operators. In fuzzy theory the membership function plays an important role, but the exact definition

of this function is often unclear. In Pliant systems we use a so-called distending function, which represents a soft inequality. In the Pliant system the various operators, which are called the conjunction, disjunction and aggregative operators, are closely related to each other. In the Pliant system we have a generator function and using this function we can create aggregation operator, conjunctive operator or disjunctive operator. In the Pliant systems the corresponding aggregative operators of the strict t-norm and strict t-conorm are equivalent, and DeMorgans law is obeyed with the corresponding strong negation of the strict t-norm or t-conorm.

The Pliant system has a strict, monotonously increasing t-norm and t-conorm, and the following expression is valid for the generator function:

$$f_c(x)f_d(x) = 1, \tag{1}$$

where $f_c(x)$ and $f_d(x)$ are the generator functions for the conjunctive and disjunctive logical operators, respectively. This system is defined in the [0,1] interval.

The operators of the Pliant system are

$$c(\mathbf{x}) = \frac{1}{1 + \left( \sum_{i=1}^{n} w_i \left( \frac{1-x_i}{x_i} \right)^{\alpha} \right)^{1/\alpha}} \tag{2}$$

$$d(\mathbf{x}) = \frac{1}{1 + \left( \sum_{i=1}^{n} w_i \left( \frac{1-x_i}{x_i} \right)^{-\alpha} \right)^{-1/\alpha}} \tag{3}$$

$$a_{\nu_*}(\mathbf{x}) = \frac{1}{1 + \left( \frac{1-\nu_*}{\nu_*} \right) \prod_{i=1}^{n} \left( \frac{1-x_i}{x_i} \frac{1-\nu_*}{\nu_*} \right)^{w_i}} \tag{4}$$

$$n(x) = \frac{1}{1 + \left( \frac{1-\nu_*}{\nu_*} \right)^2 \frac{x}{1-x}}, \tag{5}$$

$$\kappa_{\nu}^{(\lambda)}(x) = \frac{1}{1 + \frac{1-\nu_0}{\nu_0} \left( \frac{\nu}{1-\nu} \frac{1-x}{x} \right)^{\lambda}}$$

where $\nu_* \in ]0, 1[$, with generator functions

$$f_c(x) = \left( \frac{1-x}{x} \right)^{\alpha} \qquad f_d(x) = \left( \frac{1-x}{x} \right)^{-\alpha}, \tag{6}$$

where $\alpha > 0$.

The operators $c$, $d$ and $n$ fulfill the DeMorgan identity for all $\nu$, $a$ and $n$ fulfill the self-DeMorgan identity for all $\nu$, and the aggregative operator is distributive with the strict t-norm or t-conorm. The $\nu$ value express the expected value of the given context. This means that if the given $x$ value is greater than $\nu$, then the operators increase the value of $x$. The opposite is true when $x$ is smaller than $\nu$.

Table 1: Normalization parameters

| Parameter | Lambda | Shift |
|---|---|---|
| General VM cost | -1.0/96.0 | 15 |
| Cost of the application | -1.0 | (maxPrice-minPrice)/2 |
| Workload | -1.0 | maxWorkload |
| Number of VM | -1.0/8.0 | 3 |
| Number of stations | -0.125 | sumStations/appSize |
| Number of active stations | -0.125 | sumStation/activeStation |
| VM memory size | 1.0/256 | 350 |
| VM CPU | 1.0/32 | 3 |

These algorithms calculate a score for each cloud using the environment properties. The calculation step includes a normalization step, where we apply the Sigmoid function. In the normalization step it should be mentioned that if the normalized value is close to one, it means it is a more valuable property, and if the normalized value is close to zero, it means it is a less prioritized property. For example, if the CPU utilization of the VM is high, the normalization algorithm should give a value close to zero.

In a previous work [19], we used the Pliant system approach to schedule applications to VMs in a cloud by minimizing energy consumption. There we experienced that uncertainty could be well tolerated with this approach, and better results can be achieved with this model than traditional approaches.

In this work we create a new algorithm that can predict, which cloud could be the best for managing a given IoT device. This algorithm is also based on the Pliant logic, therefore for each cloud (i.e. for each VM queue in a cloud) it calculates a score number. The first step of the algorithm is to normalize the data into the [0,1] interval. We apply a Sigmoid function for this purpose. We define the following properties for each cloud VM: general VM cost, current cost of application, workload, number of running VMs in the hosting cloud, number of devices that are already connected to a cloud, memory size and number of CPUs. In Table 1 we can see the exact values of the normalization functions.

After the normalization step we modify the normalized value to emphasize the importance of the result. This means that if the given $x$ value is greater than our expectation ($\nu$) than we will increase the value of $x$. the opposite is true when the given $x$ is smaller than $\nu$. To achieve this we will modify the normalized value by using the Kappa function shown in Figure 6 with $\nu = 0.4$ and $\lambda = 3.0$ parameters:

$$\kappa_\nu^\lambda(x) = \frac{1}{1 + \left(\frac{\nu}{1-\nu}\frac{1-x}{x}\right)^\lambda} \tag{7}$$

Finally, to calculate a cloud score number for the given application. For this

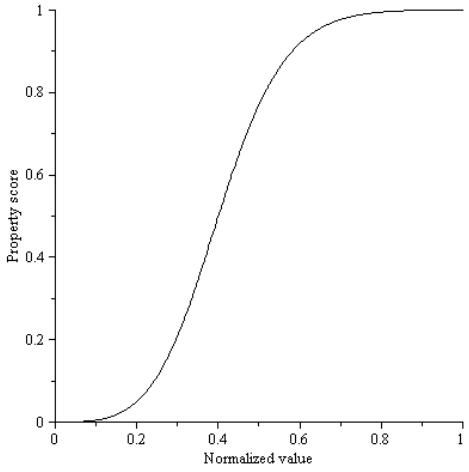



Figure 6: The Kappa function

manner we can use conjunction, disjunction or aggregation operator. The conjunctive operator is similar as the and operator. This means that if one of the value is small, then the result will be also small. The opposite is true for disjunctive operator, that is similar to or operator. If on of the value is large, the result will be also large. The aggregation operator lies between the disjunctive or conjunctive operator, that is why we use this operator:

$$a_{\nu,\nu_0}(x_1,\cdots,x_n)=\frac{1}{1+\frac{1-\nu_0}{\nu_0}\frac{\nu}{1-\nu}\prod_{i=1}^{n}\frac{1-x_i}{x_i}}, \tag{8}$$

where $\nu$ is the neutral value and $\nu_0$ is the threshold value of the corresponding negation. Here we don't want to threshold the result so both parameters have the same value 0.5. The result of the calculation is always a real number that lies in the [0,1] interval. So we calculate the score for all clouds (i.e. VM queues of clouds) to find which one is the most suitable for a given device.

# 5 Evaluation with weather forecasting scenarios

One of the earliest examples of sensor networks comes from the field of weather prediction, therefore we chose to model meteorological services based on available public information. Not only the managing architecture, but the generated sensor data is also modeled, which are in most cases: temperature, humidity, barometric pressure, rainfall and wind properties. In our model the weather conditions are regularly refreshed by the service websites in every 5 minutes, but the sensors are able to generate data in every minute, which needs caching not to overload the service. In this paper our proposed algorithms address the optimization of cloud

Table 2: Detailed Bluemix, Azure and Amazon pricing-based private cloud configurations used in the evaluations

| Cloud | Bluemix | | |
|---|---|---|---|
| Flavor | Small | Medium | Large |
| Hourly price (Euro) | 0.0378 | 0.149 | 0.295 |
| CPU (Cores)/RAM (GB) | 1/1 | 4/2 | 8/4 |
| Cloud | Azure | | |
| Flavor | Small | Medium | Large |
| Hourly price (Euro) | 0.019 | 0.0579 | 0.297 |
| CPU (Cores)/RAM (GB) | 1/1.75 | 2/3.5 | 8/14 |
| Cloud | Amazon | | |
| Flavor | Small | Medium | Large |
| Hourly price (Euro) | 0.0229 | 0.0415 | 0.3327 |
| CPU (Cores)/RAM (GB) | 1/2 | 2/4 | 8/32 |

Table 3: Detailed multi-cloud configuration for the evaluations

| Cloud | Physical machines |
|---|---|
| LPDS-1 | 1 PM - 32 cores, 128 GB RAM<br>4 PMs - 8 cores, 12 GB RAM |
| LPDS-2 | 1 PM - 64 cores, 128 GB RAM<br>1 PMs - 48 cores, 128 GB RAM<br>1 PMs - 32 cores, 128 GB RAM<br>9 PMs - 8 cores, 12 GB RAM |
| LPDS-3 | 2 PM - 64 cores, 128 GB RAM<br>2 PMs - 48 cores, 128 GB RAM<br>2 PMs - 32 cores, 128 GB RAM<br>18 PMs - 8 cores, 12 GB RAM |

side costs with enhanced allocation of the stations (i.e. devices). Which means we can define more cloud providers with their own pricing schemes, but we use only one IoT provider (therefore the IoT side cost cannot be optimized).

## 5.1 Scenario N$^o$1

In the first scenario we chose to model the crowd-sourced meteorological service of Hungary called *Idokep.hu* [21]. In this scenario we aimed to model its real-world operation: all stations have 8 sensors (represented by a device in our model), the message size of the sensors can be set up to 0.05 KBs, and the sensors generate

Table 4: Evaluation results of the scenario N$^o$1

| Strategies | Cost-aware | Random | Runtime-Aware | Pliant |
|---|---|---|---|---|
| App-1 cost | 0 | 1.119 | 1.119 | 1.119 |
| App-2 cost | 0 | 2.027 | 2.027 | 2.027 |
| App-3 cost | 0 | 16.167 | 16.223 | 16.223 |
| App-4 cost | 0 | 1.842 | 1.842 | 1.842 |
| App-5 cost | 0 | 7.300 | 7.300 | 7.300 |
| App-6 cost | 0 | 14.426 | 14.426 | 14.426 |
| App-7 cost | 1.769 | 0.974 | 0.972 | 0.974 |
| App-8 cost | 0 | 2.827 | 2.822 | 2.82 7 |
| App-9 cost | 0 | 14.478 | 14.454 | 14.478 |
| Total cost (Euro) | 1.769 | 61.164 | 61.188 | 61.219 |
| No. of used VMs | 5 | 9 | 9 | 9 |
| Total tasks | 227 | 2619 | 2616 | 2619 |
| Timeout (min) | 1.76 | 4.01 | 4.05 | 2.06 |

data in every minute. The start-up period of the stations were selected randomly between 0 and 20 min. In order to exemplify the usage of different cloud selection strategies, we defined periodic start-up and shut-down dates for certain stations (e.g. to represent malfunctions or failures). We simulate a whole day of operation (from 0:00 a.m. to 24:00 a.m.), and we start the simulation by setting up 200 stations at 0:00 a.m. At 2:00 a.m. we start 100 more, and at 10:00 a.m. 200 more to scale the total number of operated stations up to 500. At 2.00 p.m. we shut down 200 stations to scale down the number of running station to 300 by 10 p.m. At the end of the day the total number of running stations return to 200. This means the total number of operated meteorological stations in this scenario are 500 (which denotes a relatively small scale, nation-wide system).

With these station management timings we run four different test cases: (i) all stations use the random strategy, (ii) all stations use the cost-aware strategy, then (iii) all stations use the runtime-aware strategy. Finally, (iv) we used the Pliant strategy in the last experiment. For this evaluation we configured three clouds based on the *LPDS-1* cloud description from Table 3, and every cloud can run application instances (to execute compute tasks) in three VM flavors defined in Table 2 (that makes 9 possible application instances in total).

We executed the formerly defined scenario with the four test cases. The results of the experiments can be seen in Table 4. After executing this scenario the applications processed 173.75 MBs of data. The so-called timeout parameter denotes how much time it took for the application to terminate (i.e. to perform all remaining data processing operations) after the last station stopped working (at 24:00 a.m.). As we can see from these results, the cheapest solution is the cost-aware strategy (1.769 Euros) with these simulation parameters, it also has the shortest

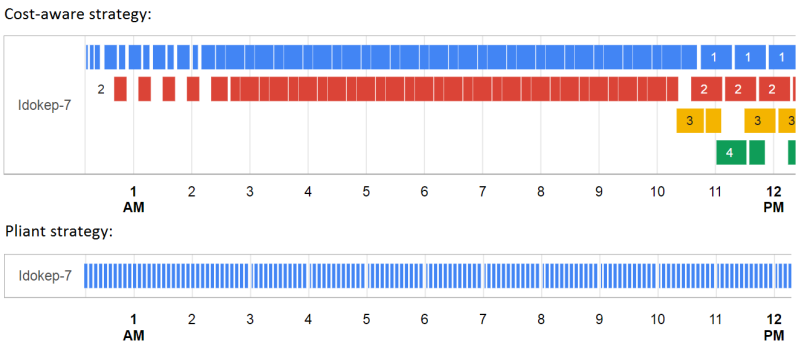



Figure 7: Timeline comparing task allocations of pliant and cost-aware strategies in Scenario N$^o$1

timeout (1.76 minutes) and it utilized the least virtual machines. This strategy only used 5 instances of the cheapest VM while the other strategies used 1 VM instance in every running application. Since the stronger virtual machines (having higher costs) processed the tasks faster than the weaker ones (as expected), they had to generate tasks more frequently. In general, choosing the cheapest VM for an application may result in serious delay in real time systems, but in this case our simulated system can operate with weaker resources in real time due to the small amount of sensor data to be processed. The beginning of the simulation there is no virtual machine running, which can serve any task execution request, thus each simulation has to wait at most 5 minutes to deploy one and allocate tasks, which means all timeout values of the strategies are acceptable.

The random, runtime-aware and Pliant strategies use unnecessarily more expensive virtual machines, which results in more than 60 Euros of cost in every case, but we can see the advantage of the pliant strategy: it tries to minimize the timeout value. In this case, it achieved a timeout of 2.06 minutes, which is the second best result. It shows that our Pliant strategy focused more on execution time reduction than cost savings.

Figure 7 shows the allocated tasks of an application running in the simulation with the pliant and cost-aware strategies for the first 12 hours. Every box denotes a different task and boxes having the same color were processed by the same virtual machine. The lengths of the tasks refers to their execution time. We can see that the tasks of the cost-aware strategy processed relatively medium amount of data resulting in many, not too narrow boxes on the timeline. In case the amount of unprocessed data was growing, the system started to scale up the number of utilized virtual machines. Meanwhile the pliant strategy worked with stronger and faster virtual machines, which resulted also in many tasks with small amount of data, but using only the same, best fit VM. This explains the difference between the number of total tasks of these strategies. Next we investigate a scenario of a higher scale.

Table 5: Evaluation results of the scenario N$^o$2

| Strategies | Cost-aware | Random | Runtime-Aware | Pliant |
|---|---|---|---|---|
| App-1 cost | 0 | 3.563 | 3.544 | 3.542 |
| App-2 cost | 0 | 3.745 | 3.707 | 3.721 |
| App-3 cost | 0 | 12.396 | 12.451 | 12.451 |
| App-4 cost | 0 | 5.799 | 5.796 | 5.783 |
| App-5 cost | 0 | 9.384 | 8.324 | 8.748 |
| App-6 cost | 0 | 12.157 | 12.132 | 12.034 |
| App-7 cost | 26.419 | 3.061 | 3.063 | 3.090 |
| App-8 cost | 0 | 5.156 | 5.243 | 5.166 |
| App-9 cost | 0 | 11.261 | 11.187 | 11.112 |
| Total cost (Euro) | 26.419 | 66.527 | 65.451 | 65.651 |
| No. of used VMs | 109 | 180 | 170 | 173 |
| Total tasks | 1722 | 1830 | 1819 | 1838 |
| Timeout (min) | 631 | 86 | 86 | 71 |

## 5.2 Scenario N$^o$2

In the second scenario we aimed to simulate a larger, world-wide system. An international meteorological service called OpenWeatherMap [22] is operated by the Openweather IT company [23], which was established in 2014 by a group of experts in Big Data and image processing. As their website suggests, they manage over 40000 meteorological stations all over the world. Our goal with this scenario is to investigate how IoT applications behave in such large-scale environments. Similarly to the first scenario, we used three clouds configured with Amazon, Azure and IBM Bluemix cloud provider pricing defined in Table 2, but we modified the physical parameters of the simulated private clouds (to be able to cope with the higher number of stations) as defined by *LPDS-2* in Table 3. The number of running weather stations has been increased to 40000, each of them works with 8 sensors and generate 50 bytes of data every minute. We run this scenario to simulate 6 working hours. In the beginning we started 10000 stations, then we added 10000 stations more in the next hours to reach 40000 stations by the fourth hour.

The results of the second scenario is shown in Table 5. After executing this scenario the applications processed 4.008 GBs of data. In the previous scenario the cost-aware strategy was good choice both cost and runtime, but problems may occur for systems with higher scale. In this case the cheapest schedule was provided by the cost-aware strategy with 26.419 Euros, but it had 631 minutes ($\sim$10.51 hours) timeout, which is almost twice longer than the simulated working time (i.e. 6 hours). For applications which are not sensitive to low latency, the cost-aware strategy can be still an acceptable opportunity to decrease costs, but for time-dependent applications (e.g. smart systems, weather forecasting systems) other strategies are

Table 6: Evaluation results of the scenario N$^o$3

| Strategies | Cost-aware | Random | Runtime-Aware | Pliant |
|---|---|---|---|---|
| App-1 cost | 0 | 3.512 | 3.552 | 3.552 |
| App-2 cost | 0 | 3.724 | 3.731 | 3.804 |
| App-3 cost | 0 | 13.283 | 12.479 | 12.451 |
| App-4 cost | 0 | 6.233 | 5.830 | 5.824 |
| App-5 cost | 0 | 9.197 | 8.523 | 8.386 |
| App-6 cost | 0 | 12.428 | 12.157 | 11.960 |
| App-7 cost | 26.489 | 3.085 | 3.071 | 3.070 |
| App-8 cost | 0 | 5.224 | 5.185 | 5.195 |
| App-9 cost | 0 | 11.904 | 12.251 | 12.152 |
| Total cost (Euro) | 26.489 | 66.429 | 66.784 | 66.397 |
| Used VMs | 172 | 183 | 184 | 185 |
| Total tasks | 1722 | 1905 | 1889 | 1893 |
| Timeout (min) | 526 | 36 | 36 | 36 |

needed. Nevertheless, the cost-aware strategy utilized the lowest number of VMs, too. The random and the runtime-aware strategies have the same timeout (with 86 minutes), but the runtime-aware approach operated with less virtual machines (with 10 VMs) and saved around one Euro compared to the random one. The pliant strategy was even better with almost the same price, since it reached the most favorable timeout (with 71 minutes).

## 5.3 Scenario N$^o$3

In the third scenario we configured our private cloud to be the strongest, having twice as many resources as in the second scenario (detailed in the *LPDS-3* parameter setup of Table 3), while the rest of the configuration (the applications and the stations) remained untouched, thus the final amount of generated (and processed) data was the same as in the previous scenario.

Table 6 shows the results of the third scenario. With the increased physical resources the running time have decreased, but the cost-aware strategy still required 526 minutes (∼8.76 hours) timeout, after the last station stopped working.

If we take a look at the figures, we can see that most strategies benefited from the stronger clouds: they all managed to reduce the timeout significantly. The cost-aware strategy remained the cheapest one, but the number of used virtual machines increased the most against the other strategies compared with the second scenario. The amount of unprocessed data grew faster, than the number of available virtual machines, thus when the application operated with the maximum number of stations the stronger resources could provide more virtual machines to reduce timeout. Comparing the other three strategies, it shows minimal deviation in the

Table 7: Evaluation results of the scenario N$^o$4

| Strategies | Cost-aware | Runtime-Aware | Pliant |
|---|---|---|---|
| Total cost (Euro) | 10.442 | 41.765 | 38.84 |
| Used VMs | 81 | 51 | 51 |
| Total tasks | 685 | 1384 | 1242 |
| Timeout (min) | 41 | 31 | 24.9 |

used virtual machines or the costs. The pliant approach uses the most virtual machines (185), but it was the cheapest with 66.397 Euros, but all strategies has the same timeout value with 36 minutes. This means that by increasing the number of resources, the strategies behave differently.

### 5.4 Scenario N$^o$4

In the previous scenarios the station allocation strategies had to choose only 2-4 times to select VM-queues for the applications processing sensor data of the stations. One of advantages of the pliant approach is that it is able to take into account more features of the underlying systems, but for this strategy these scenarios were too static, having only a small number of decision points. Thus in the last, fourth scenario we defined a more dynamic scenario, where we managed 11500 stations in the following way. Every half an hour, 500 stations were started to operate and the whole simulation run for 12 hours. The pliant algorithm had to decide more often than in the former cases. Our aim with this scenario is to prove that this sophisticated algorithm is able to decrease both the costs and the runtime at the same time. The results can be seen in Figure 7. The processed data for the whole experiment is 1.54 GBs. For this scenario we used a different cloud setup as well. We configured three clouds based on the *LPDS-1*, *LPDS-2* and *LPDS-3* cloud description of Table 3, respectively.

As expected the cheapest solution is the Cost-aware algorithm with 10.442 Euros, which also has the highest timeout with 41 minutes. This strategy used the highest number of virtual machines, which is also a disadvantage, if the cloud provider calculates the cost based on the number of VMs. Comparing the other strategies (here we neglected the random approach), the pliant and the runtime-aware strategies used the same number of virtual machines, but the Pliant algorithm managed to reduce both the cost and the runtime most effectively.

## 6 Conclusion

Cloud Computing solutions act as supporting services for the IoT world. Applications in this newly emerged field are continuously growing, and further research is

still needed to resolve open issues, to optimize system management and to reduce utilization costs for both providers and end-users.

In this paper we introduced four cloud selection strategies aimed to reduce IoT application execution time and usage costs. We evaluated these strategies through scenarios derived from a real-life weather forecasting service. The results have shown that we can achieve significant cost savings or makespan reduction in multi-cloud systems by using one of our proposed strategies.

Our investigations showed that if the components of IoT-Cloud systems (including sensors, smart devices and virtual resources) change often, a static scheduling and placement algorithm (ignoring the actual load, the type of virtual machines and the number of physical resources) can provide increased latencies and costs. Our presented a dynamic approach based on the Pliant method can adapt to the actual state of the underlying, possibly multi-cloud systems, therefore it can find better placement of devices resulting in lower costs and response times.

Our future work will address IoT scenarios from other smart domains (e.g. smart farming), as well as the modeling of additional sensor and device types. We also plan to extend our cloud selections algorithms to minimize IoT side costs, and to introduce energy-aware algorithms for smart device management. We also plan to perform experiments with models of recent infrastructures (e.g. Agrodat [25], MTA Cloud [26] or the Cloud for Education [27]).

The presented scenarios and the source code of the IoT extended DISSECT-CF with the mentioned XML description formats and the XML Schema Document files are available at [20].

## 7 Acknowledgement

This work was supported by the Hungarian Government under the grant number EFOP-3.6.1-16-2016-00008.

## References

[1] A. Botta, W. de Donato, V. Persico, and A. Pescape. Integration of Cloud computing and Internet of Things. Future Gener. Comput. Syst. 56, pp. 684-700, 2016.

[2] G. Kecskemeti. DISSECT-CF: A simulator to foster energy-aware scheduling in infrastructure clouds. *Simul. Model. Pract. Theory*, 58P2, 2015.

[3] A. Markus, A. Kertesz, G. Kecskemeti. Cost-aware iot extension of dissect-cf. *Future Internet*, 9(3), 2017.

[4] A. Kertesz. Characterizing cloud federation approaches. In: Cloud computing: challenges, limitations and R&D solutions. Computer communications and networks. Springer, Cham, pp. 277-296, 2014.

[5] G. Gultekin Varkonyi, Sz. Varadi, A. Kertesz. Legal Aspects of Operating IoT Applications in the Fog. In: Fog and Edge Computing: Principles and Paradigms. John Wiley & Sons, Hoboken, pp. 411-432, 2019.

[6] Calheiros, R.N.; Ranjan, R.; Beloglazov, A.; De Rose, C.A.; Buyya, R. CloudSim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms. *Software: Practice and Experience*, *41*, 23–50, 2011.

[7] Khan, A.M.; Navarro, L.; Sharifi, L.; Veiga, L. Clouds of small things: Provisioning infrastructure-as-a-service from within community networks. Wireless and Mobile Computing, Networking and Communications (WiMob), 2013 IEEE 9th International Conference on. IEEE, pp. 16–21, 2013.

[8] H. Gupta, A. V. Dastjerdi, S. K. Ghosh, R. Buyya. iFogSim: A toolkit for modeling and simulation of resource management techniques in the Internet of Things, Edge and Fog computing environments. Softw. Pract. Exper. 47:12751296, 2017.

[9] Sotiriadis, S.; Bessis, N.; Asimakopoulou, E.; Mustafee, N. Towards simulating the Internet of Things. 28th International Conference on Advanced Information Networking and Applications Workshops (WAINA), pp. 444–448, 2014.

[10] Sotiriadis, S.; Bessis, N.; Antonopoulos, N.; Anjum, A. SimIC: Designing a new Inter-Cloud Simulation platform for integrating large-scale resource management. IEEE 27th International Conference on Advanced Information Networking and Applications (AINA), pp. 90–97, 2013.

[11] Moschakis, I.A.; Karatza, H.D. Towards scheduling for Internet-of-Things applications on clouds: a simulated annealing approach. *Concurrency and Computation: Practice and Experience*, *27*, 1886–1899, 2015.

[12] Silva, I.; Leandro, R.; Macedo, D.; Guedes, L.A. A dependability evaluation tool for the Internet of Things. *Computers & Electrical Engineering*, *39*, 2005–2018, 2013.

[13] Zeng, X.; Garg, S.K.; Strazdins, P.; Jayaraman, P.P.; Georgakopoulos, D.; Ranjan, R. IOTSim: A simulator for analysing IoT applications. *Journal of Systems Arch.*, 2016.

[14] A. Kertesz, T. Pflanzner, T. Gyimothy. A Mobile IoT Device Simulator for IoT-Fog-Cloud Systems. Journal of Grid Computing, 2018. 10.1007/s10723-018-9468-9.

[15] T. Pflanzner, A. Kertesz. A Taxonomy and Survey of IoT Cloud Applications. EAI Endorsed Transactions on Internet of Things, 2018.

[16] J. Dombi. A general class of fuzzy operators, the de morgan class of fuzzy operators and fuzziness measures induced by fuzzy operators. Fuzzy Sets and Systems 8, 1982.

[17] Yager, Ronald R. and Zadeh, Lotfi A. An Introduction to Fuzzy Logic Applications in Intelligent Systems Kluwer Academic Publishers, 0792391918, 1992

[18] J. Dombi. Pliant system. IEEE International Conference on Intelligent Engineering System Proceedings, Budapest, Hungary, 1997.

[19] A. Kertesz, J. D. Dombi, A. Benyi. A Pliant-based Virtual Machine Scheduling Solution to Improve the Energy Efficiency of IaaS Clouds. Journal of Grid Computing, Vol. 14, pp. 41–53, 2016.

[20] DISSECT-CF extensions towards IoT. Online: https://github.com/andrasmarkus/dissect-cf/tree/pricing/. Accessed in September, 2018.

[21] Websize of Idokep.hu. Online: http://idokep.hu. Accessed in August, 2017.

[22] OpenWeatherMap Station information site. Online: https://openweathermap.org/stations-old. Accessed in September, 2018.

[23] OpenWeather company website. Online: https://openweather.co.uk/about. Accessed in September, 2018.

[24] LPDS Cloud of MTA SZTAKI. Online: https://www.sztaki.hu/tudomany/reszlegek/lpds. Accessed in September, 2018.

[25] R. Lovas, K. Koplanyi, G. Elo. Agrodat: A Knowledge Centre and Decision Support System for Precision Farming Based on IoT and Big Data Technologies. Special theme: Smart Farming, ERCIM News 113. April, 2018.

[26] MTA Cloud website. Online: https://cloud.mta.hu. Accessed in January, 2019.

[27] Cloud for Education website of KIFU. Online: http://kifu.gov.hu/szolgaltatasok/ ikt/felho/cloud_for_education. Accessed in January, 2019.

Acta Cybernetica **24** (2019) 105–120.

# Different Types of Search Algorithms for Rough Sets*

Dávid Nagy[a], Tamás Mihálydeák[b], and László Aszalós[c]

**Abstract**

Based on the available information in many cases, it can happen that two objects cannot be distinguished. If a set of data is given and in this set two objects have the same attribute values, then these two objects are called indiscernible. This indiscernibility has an effect on the membership relation because in some cases it makes our judgment uncertain about a given object. The uncertainty appears because if something about an object needs to be stated, then all the objects that are indiscernible from the given object must be taken into consideration. The indiscernibility relation is an equivalence relation which represents the background knowledge embedded in an information system. In a Pawlakian system this relation is used in set approximation. Correlation clustering is a clustering technique which generates a partition. In the authors' previous research, (see in [10, 11, 9]) the possible usage of correlation clustering in rough set theory was investigated. In this paper, the authors show how different types of search algorithms can affect the set approximation.

**Keywords:** rough set theory, set approximation, data mining

## 1 Introduction

Pawlak's indiscernibility relation (which is an equivalence relation) represents a limit of our knowledge embedded in an information system. This relation defines the base sets. They contain objects that are indiscernible from each other. In many applications, it is common to replace the equivalence relation with tolerance relation. In our previous study, we examined whether the clusters, generated by correlation clustering, can be understood as a system of base sets. Correlation clustering is a clustering method in data mining which creates a partition based on a tolerance relation. The groups, defined by this partition, contain similar

*This work was supported by the National Research, Development and Innovation Office of Hungary under Grant No. TÉT 16-1-2016-0193.

[a]Faculty of Informatics, University of Debrecen, E-mail: {nagy.david}@inf.unideb.hu

[b]Faculty of Informatics, University of Debrecen, E-mail: {mihalydeak.tamas}@inf.unideb.hu

[c]Faculty of Informatics, University of Debrecen, E-mail: {aszalos.laszlo}@inf.unideb.hu

DOI: 10.14232/actacyb.24.1.2019.8

objects. In our previous papers, we showed that it is worth to generate the system of base sets from the partition. This way, the base sets contain objects that are typically similar to each other and they are pairwise disjoint. To find the partition in reasonable time, search algorithms must be used. So the system of base sets is highly dependent on the used search algorithm. The structure of the paper is the following: A theoretical background about the classical rough set theory comes first. In section 3 we define correlation clustering mathematically. In section 4 we present our previous work. In section 5 we present the search algorithm used in our experiments. In section 6 we show a way to compare the algorithms. Finally we conclude the results.

## 2 Theoretical Background

In practice, a set is a collection of objects that are similar in some sense. A set is uniquely identified by its members. It means that if we would like to decide, whether an object belongs to this set, then we can give a precise answer which is yes or no. For instance, the set of even numbers is a crisp set because it can be decided if an arbitrary number is even or odd. However, in some computer science applications, researchers are interested in vague concepts. The notion of a brave warrior is vague because we cannot give two disjoint classes: brave and not brave warriors. One can consider a person brave due to their actions, but maybe someone else would consider this person as not brave. So bravery is a vague concept.

Rough set theory was proposed by professor Pawlak in 1982 (see in [12]). The theory offers a way to handle vague concepts. Each object of a universe can be described by a set of attribute values. If two objects have the same known attribute values, then these objects cannot distinguished. The indiscernibility relation generated in this way is the mathematical basis of rough set theory.

If we want to decide, whether an object belongs to an arbitrary set, based on the available data, then our decision affects the decision about all the objects that are indiscernible from the given object.

In this case, if we would like to check, whether an object is in an arbitrary set, then the following three possibilities appear:

- it is sure that the object is in the set if all the objects, that are indiscernible from the given object, are in the set;
- the object may be in the set if there some objects that are in the set and are indiscernible from the given object;
- it is sure that the object is not in the set if all the objects, that are indiscernible from the given object, are not in the set.

So the indiscernibility makes a set vague.

From the theoretical point of view, a Pawlakian approximation space (see in [12, 13, 14]) can be characterized by an ordered pair $\langle U, \mathcal{R} \rangle$ where $U$ is a nonempty set

of objects and $\mathcal{R}$ is an equivalence relation on $U$. In order to approximate an arbitrary subset $S$ of $U$ the following tools have to be introduced:

- *the set of base sets*: $\mathfrak{B} = \{B \mid B \subseteq U,$ and $x, y \in B$ if $x\mathcal{R}y\}$, the partition of $U$ generated by the equivalence relation $\mathcal{R}$;
- *the set of definable sets*: $\mathfrak{D}_{\mathfrak{B}}$ is an extension of $\mathfrak{B}$, and it is given by the following inductive definition:
  1. $\mathfrak{B} \subseteq \mathfrak{D}_{\mathfrak{B}}$;
  2. $\emptyset \in \mathfrak{D}_{\mathfrak{B}}$;
  3. if $D_1, D_2 \in \mathfrak{D}_{\mathfrak{B}}$, then $D_1 \cup D_2 \in \mathfrak{D}_{\mathfrak{B}}$.
- *the functions* $\mathsf{l}, \mathsf{u}$ form a Pawlakian approximation pair $\langle \mathsf{l}, \mathsf{u} \rangle$, i.e.
  1. $Dom(\mathsf{l}) = Dom(\mathsf{u}) = 2^U$
  2. $\mathsf{l}(S) = \bigcup\{B \mid B \in \mathfrak{B}$ and $B \subseteq S\}$;
  3. $\mathsf{u}(S) = \bigcup\{B \mid B \in \mathfrak{B}$ and $B \cap S \neq \emptyset\}$.

$U$ is the set of objects. $\mathfrak{B}$ is the system of base sets which represents the background knowledge. $\mathfrak{D}_{\mathfrak{B}}$ is the set of definable sets which defines how the base sets can be used in the set approximation. The functions $l$ and $u$ give the lower and upper approximation of a set. The lower approximation contains objects that surely belong to the set, and the upper approximation contains objects that possibly belong to the set. The set $BN(S) = u(S) \setminus l(S)$ is called the boundary region of the set $S$. If $BN(S) = \emptyset$ then $S$ is crisp, otherwise it is rough.

Table 1 shows a very simple table containing 8 rows. Each of them represents a patient and each has 3 attributes: headache, body temperature and muscle pain. The base sets contain patients with the same symptoms (which means they are indiscernible from each other) and it is the following:

$$\mathfrak{B} = \{\{u_1\}, \{u_2\}, \{u_3\}, \{u_4\}, \{u_5, u_7\}, \{u_6, u_8\}\}$$

Let us suppose that based on some background knowledge we know that the patients $u_1$, $u_2$ and $u_5$ have the flu. Let $S$ be the following set of these patients: $\{u_1, u_2, u_5\}$. The approximation of the set $S$ is the following:

- $l(S) = \{\{u_1\}; \{u_2\}\}$
- $u(S) = \{\{u_1\}; \{u_2\}; \{u_5, u_7\}\}$
- $BN(S) = \{\{u_5, u_7\}\}$

Here, $l(S)$ contains those patients that surely have the flu. Patient $u_5$ is not in the lower approximation because there is one other patient, $u_7$, who is indiscernible from $u_5$, and we do not have information about, whether $u_7$ has the flu or not. So the base set $\{u_5, u_7\}$ can only be in the upper approximation.

Table 1: Information System

| Object | Headache | Body Temp. | Muscle Pain |
|---|---|---|---|
| $u_1$ | YES | Normal | NO |
| $u_2$ | YES | High | YES |
| $u_3$ | YES | Very high | YES |
| $u_4$ | NO | Normal | NO |
| $u_5$ | NO | High | YES |
| $u_6$ | NO | Very high | YES |
| $u_7$ | NO | High | YES |
| $u_8$ | NO | Very High | YES |

# 3 Correlation Clustering

Data mining is the process of discovering patterns and hidden information in large data sets. The goal of a data mining process is to extract information from a data set and transform it into an understandable structure for further use. Clustering is a data mining technique in which the goal is to group objects, so that the objects in the same group are more similar to each other than to those in other groups. In many cases, the similarity is based on the attribute values of the objects. In most of them, some kind of distance is used to define the similarity. However, sometimes only nominal data are given. In this particular case, distance can be meaningless. For example, what is the distance between a male and a female? In this case, a similarity relation can be used which is a tolerance relation. If this relation holds for two objects, we can say that they are similar. If this relation does not hold, then they are dissimilar. It is easy to prove that this relation is reflexive and symmetric. The transitivity; however, does not hold necessarily. Correlation clustering is a clustering technique based on a tolerance relation (see in [3, 4, 17]).

Let $V$ a set of objects and $T$ the similarity relation. The task is to find an $R \subseteq V \times V$ equivalence relation which is *closest* to the tolerance relation.

A (partial) tolerance relation $T$ (see in [15, 8]) can be represented by a matrix $M$. Let matrix $M = (m_{ij})$ be the matrix of the partial relation $T$ of similarity:

$$m_{ij} = \begin{cases} 1 & i \text{ and } j \text{ are similar} \\ -1 & i \text{ and } j \text{ are different} \\ 0 & \text{otherwise} \end{cases}$$

A relation is called partial if there exist two elements $(i, j)$ such that $m_{ij} = 0$. It means that if we have an arbitrary relation $R \subset V \times V$ we have two sets of pairs. Let $R_{true}$ be the set of those pairs of elements for which the $R$ holds and $R_{false}$ be the one for which $R$ does not hold. If $R$ is partial, then $R_{true} \cup R_{false} \subset V \times V$. If $R$ is total, then $R_{true} \cup R_{false} = V \times V$.

A partition of a set $S$ is a function $p : S \to \mathbb{N}$. Objects $x, y \in S$ are in the same cluster at partitioning $p$, if $p(x) = p(y)$. We treat the following two cases conflicts for any $x, y \in V$:

- $(x, y) \in T$ but $p(x) \neq p(y)$
- $(x, y) \notin T$ but $p(x) = p(y)$

The goal is to minimize the number of these conflicts. If their number is 0, the partition is called *perfect*. Given the $T$ and $R$, we call the number of conflicts the distance of the two relations. The partition given this way, generates an equivalence relation. This relation can be considered as the closest to the tolerance relation.

The number of partitions can be given by the Bell number (see in [1]) which grows exponentially. For more than 15 objects, we cannot achieve the optimal partition by exhaustive search in reasonable time. In a practical case, a search algorithm can be used which can give a quasi-optimal partition.

# 4 Similarity based rough sets

In practical applications, indiscernibility relation is too strong. Therefore, Pawlakian approximation spaces have been generalized using tolerance relations (symmetric and reflexive) which are similarity relations. Covering-based approximation spaces (see [16]) generalize Pawlakian approximation spaces in two points:

1. $\mathcal{R}$ is a tolerance relation;
2. $\mathfrak{B} = \{[x] \mid [x] \subseteq U, x \in U \text{ and } y \in [x] \text{ if } x\mathcal{R}y\}$, where $[x] = \{y \mid y \in U, x\mathcal{R}y\}$.

The definitions of definable sets and approximation pairs are the same as before. In these covering systems, each object generates a base set.

Correlation clustering defines a partition. The clusters contain objects that are typically similar to each other. In our previous work ([10]), we showed that this partition can be understood as the system of base sets which results in a completely new approximation space. The approximation space also has several good properties. The most important one is that it focuses on the similarity (the tolerance relation) itself, and it is different from the covering type approximation space relying on the tolerance relation.

Singleton clusters represent very little information because the system could not consider its member similar to any other objects without increasing the number of conflicts. As they mean little information, we can leave them out. If we do not consider the singleton clusters, then we can generate a partial system of base sets from this partition where singleton clusters are not base sets (see in [11, 9]).

In reasonable time, correlation clustering can only be solved by using search algorithms. However, each algorithm can provide different clusters. So the system of base sets can also be different. It is a natural question to ask, how the search algorithms can affect the structure of the base sets. As the approximation based on correlation clustering is a completely new way of approximating sets, it is crucial to use the best possible search algorithm.

# 5 Algorithms

Between 2006 and 2016, Advanced Search Methods was a compulsory subject for some Computer Science master students. Initially, students learned about the well known NP-hard problems (SAT, NLP, TSP, etc.) and various popular optimization methods. Later, to help some physicists from University Babes-Bolyai in Cluj, one co-author began to research the problem of correlation clustering. This problem can easily be formulated (which equivalence relation is the closest to a given tolerance relation?), can be quickly understood, is freely scalable, but NP-hard, and if we have more than 15 objects in a general case we can only provide an approximate solution. That is why this problem got central role from 2010. In this year, the students with the co-author's lead implemented the learned algorithms, and used correlation clustering to test and compare them. There were several didactic goals of this development: they worked as a team, where the leader changed from algorithm to algorithm, whose duty was to distribute the subtasks among the others and compose/finalize their work. The implementations together gave thousand of LOCs, so the students got experience with a real-life size problem. When designing this system as a framework, the OOP principles were of principal importance, to be able to apply it for other optimization problems. The system was completed with several refactorisations and extended with methods developed directly for correlation clustering, and several special data structures which allowed to run programs several magnitudes faster. Finally the full source of the whole system with detailed explanations was published at the Hungarian Digital Textbook Repository [2]. According to the students' request the system was written in Java. Jason Brownlee published a similar book using Ruby [5].

The following list shows the used algorithms in our experiments. Each of them can be downloaded from [2]:

- Hill Climbing Algorithm
- Stochastic Hill Climbing Algorithm
- Tabu Search
- Simulated Annealing
- Parallel Tempering
- Genetic Algorithm
- Bees Algorithm
- Particle Swarm Optimization
- Firefly Algorithm

In the next subsections, there are some brief information about the used algorithms and their parameters. The whole descriptions can also be seen in [2].

## 5.1 Hill Climbing Algorithm

This method is very well-known. Each state in the search plane represents a partition. A state is considered better than another state if its number of conflicts is less than that of the other state. In each step it is checked, whether there is a better state in the neighborhood of the actual state. If there is not, then the algorithm stops. If there is, then the next step goes from this point.

## 5.2 Stochastic Hill Climbing Algorithm

The original hill climbing search is greedy, it always moves to the best neighbor. Stochastic hill climbing is a variant, where the algorithm chooses from the neighbors in proportion to their goodness, allowing the algorithm to move in a worse direction as well.

## 5.3 Tabu Search

Each state in the search plane represents a partition like in the previous algorithms. The tabu search defines a list of banned states or directions to where it cannot move at a time. This is called a tabu list or memory. There are many types of memories. In our experiments, we used a short-term memory with the size of 50.

The neighborhood of the actual state consists of banned and permitted states. In each step, there are two possibilities:

- If one of the neighbors is so good that it is better than the best state so far, then it should go there even if it is banned. This is called the aspirant condition.
- The algorithm moves to the best permitted neighbor of the actual state.

If the new state is better than the best state so far, then this state will be the new best. The previous state will also be added to the tabu list, so the algorithm could not go backwards immediately. If the list is full, then the last state will be deleted. The algorithm stops if it reaches 1000th step and it returns the best state.

## 5.4 Simulated Annealing

Each state in the search plane represents a partition. In each step, the algorithm chooses a neighbor of the actual state. Let $f$ denote the number of conflicts in the actual state and $f'$ denote the number of conflicts in the neighbor. If $f' < f$, then the algorithm moves to the neighbor. If not, then it chooses this state with the probability of $\frac{e^{f-f'}}{T}$. The value $T$ is the temperature value which is a crucial parameter. It should decrease in each step. Determining the starting temperature value is a hard task. The common method for the issue is heating. In each temperature (starting from 1), 500 attempts are made to move to a neighbor, and the number of successful movements are counted. If the ratio of the number of

successful movements and the number of attempts reaches 0.99, then the heating procedure stops, otherwise the temperature value is increased. After the heating, the annealing (search) step comes. It is important that, how much time the algorithm spends in each temperature value. A minimal step count were defined and it increases each time the temperature is decreased, until it reaches a maximal value when the algorithm stops and returns the best state. The minimal step count was set to 100 and the maximal was set to 1000. The temperature values are always decreased by 97%.

## 5.5 Parallel Tempering

The simulated annealing runs on a single thread. This is parallelized version, where threads can cooperate with each other. In this method, we used 3 threads.

## 5.6 Genetic Algorithm

In this algorithm, each partition is represented by an entity. In each search step, there is a population of entities with the size of 100. In the beginning, each entity represents a random partition. This population contains the actual generation of entities and best entities from the old generation. In each step, the best 25 entities stay in the population. The rest of the spaces are filled with the descendants of the entities of the old generation. In step 1, the algorithm defines the new generation. Step 2 is the reproduction step. A descendant is created in this step with the crossover of 2 parent entities. In this paper, we have used one-point crossover. For the crossover, not a random or the best element is chosen but an element from a set defined by a parameter. The size of this set was 4. After step 2, each child entity goes through a mutation phase (step 3) whose probability was 2/3. After each child entity is created, the actual generation is overwritten by the new generation, and the algorithm goes back to the step 1. The algorithm stops when it reaches the 2000th generation and returns the best entity of the population.

## 5.7 Bees Algorithm

This algorithm is based on the society of honey bees. Each partition is represented by a "bee". There are two types of bees: scout bees and recruit bees. Scout bees scout the area and they report back to the hive about their findings. Then the necessary number (in proportion to the goodness of the finding) of recruit bees go to the area to forage. In our case, the scouts are scattered across the search plane and recruit bees were assigned only to the best of them. These bees are called elites. The rest of the scout bees wander in the plane. It changes dynamically which scout bees are considered as elite and how many recruits are assigned to them. The recruits search around the elite bee to which they were assigned, and if they find a better state, then the scout bees move to that position. In our experiments, the number of scout bees was set to 50 and the number of elites was 5 and 1000 recruit bees follow the elites. In the beginning, the scout bees start from a random

position. The algorithm stops when it reaches the 2000th step and it returns the partition represented by the best bee.

### 5.8 Particle Swarm Optimization

In this algorithm, each partition is represented by an insect (particle). Each insect knows its best position and the best position of the swarm. The size of the swarm was set to 50. In each step, each insect moves in the search plane. In the beginning, the insects start from a random position. There are 3 possibilities for them to move:

- Randomly move
- Move towards its best position
- Move toward the best position of the swarm

The possibilities of the moves was set to 0.2, 0.3, 0.5 respectively. After reaching the 6000th step, the algorithm stops and returns the insect with the best position.

### 5.9 Firefly Algorithm

In this algorithm, each partition is represented by a firefly. The fireflies are unisex and their brightnesses are proportionate to the goodness of the partition they represent. In the beginning, the fireflies start from a random position, and in each step each firefly moves to its brightest neighbor. If the brightest neighbor of a firefly is itself, then it moves randomly. Brightness is dependent on the distance of the insects. The intensity of a firefly is defined by the following formula: $I_d = \frac{I_0}{1+\gamma d^2}$, where $I_0$ denotes the starting intensity, $\gamma$ is the absorption coefficient (was set to 0.03) and $d$ is the distance between the two fireflies. After 10000 steps the algorithm stops, and the result is the partition which is represented by the brightest firefly. The number of fireflies was set to 50.

## 6 Comparing Algorithms

To compare the algorithms, we calculated the following values:

- Number of singleton clusters
- Standard deviation of the base set sizes
- Interquartile range of the base set sizes
- Execution time of the algorithm

In this paper, the authors refer to the cardinality of a base set as its size. As previously mentioned, singleton clusters mean little information. The greater their number is, the more unclear our knowledge becomes. For a search algorithm, the

most optimal is, if it provides the least number of these clusters in order to have a precise knowledge of the system.

The sizes of the base sets are also worth to be checked. For set approximations it is more suitable, if the sizes do not vary much. So the standard deviation of the base set sizes should be minimized as well as the interquartile range of the base set sizes.

An important parameter is the execution time of the search algorithms. It is especially crucial when there are a huge number of objects.

Most of the algorithms have many parameters, and changing them can result in different outputs. Many possible combinations were tried during our research. Dozens of tables were generated and these tables are not present in this paper, but they can be downloaded from the following link: `https://bit.ly/2sO4UoD`

For comparing the parameters, the same graph (with 100 points, q = 0.6) were used for each algorithm. Each algorithm was run three times to exclude the randomness. In each case, the optimal parameter combination was the one which minimized the above mentioned 4 values. If the differences between the standard deviations, the interquartile ranges and the numbers of singletons were not significant, then the judgment was made by the execution time.

# 7 Results

## 7.1 Erdős-Rényi graphs

**Algorithm 1** Erdős-Rényi random graph generating method

**Procedure** ER($N, p$)

```
for i = 1, ..., N do
  for j = 1, ..., N do
    Generate a random x value between 0 and 1
    if x < p then
      There is an edge between objects i and j
    end if
  end for
end for
```

In this part our experiments, we used Erdős-Rényi graphs (see in [7, 6]). This random graph generating method is very simple. Its pseudo-code can be seen in Algorithm 1. In our experiments, we used $p = 0.5$ , $p = 0.6$ and $p = 0.7$. Half of the generated edges denoted the similarity between the two objects and half of them the difference. The graphs are only used for defining a similarity relation. Any other kinds of graphs can be used. 100, 200, 300 and 400 points were generated. For each test case, each algorithm was run 3 times on the same graphs, then the averages of the values, described in section 6, were determined. The results can

Table 2: Average standard deviations of the base set sizes for Erdős-Rényi graphs

| **Points** | **HC** | **SHC** | **TABU** | **SA** | **PT** | **GE** | **BEE** | **PSO** | **FF** |
|---|---|---|---|---|---|---|---|---|---|
| **200 q=0.5** | 37 | 31 | 32 | 13 | 12 | 40 | 15 | 34 | 27 |
| **400 q=0.5** | 63 | 61 | 51 | 15 | 24 | 82 | 21 | 66 | 49 |
| **200 q=0.6** | 37 | 35 | 37 | 18 | 19 | 39 | 15 | 35 | 31 |
| **400 q=0.6** | 69 | 63 | 61 | 21 | 23 | 83 | 26 | 65 | 49 |
| **200 q=0.7** | 37 | 31 | 26 | 18 | 20 | 38 | 15 | 39 | 30 |
| **400 q=0.7** | 68 | 61 | 66 | 14 | 19 | 78 | 18 | 63 | 55 |

be seen in the following tables and can be downloaded from the following link: `https://bit.ly/2sOqMA6`.

From Table 2 the average standard deviations of the base set sizes can be read. Even for a small number of points, the differences are apparent. The simulated annealing provided the best result in most cases. Its parallel version has almost the same output. The bees algorithm also returned a rather acceptable result.

In Table 3 the distance of the sizes of the biggest and the smallest base sets can be seen. The values show almost the same tendency as in the last table. The simulated annealing, the parallel tempering and the bees algorithm proved to be the most acceptable. For 400 points, the other the algorithms provided twice or three times as large values as the other 3 which is not suitable.

In Table 4 the average numbers of singletons are listed. Here, the differences are not so significant as before. The number of points does not affect it very much.

In Table 5 the average run-time of the algorithms can be seen in seconds. The values here vary the most. It is obvious that the simulated annealing was the least affected by the increasing number of points. For 200 points, the hill climbing algorithm and its stochastic version provided the fastest output. Although, as soon as the number of points was increased, they could not compete with the simulated annealing. For 400 points, the simulated annealing was more than 20-35 times faster than the other two. The particle swarm optimization was the slowest of all the algorithms. For a huge number of points, it is basically pointless to be used.

## 7.2 Random two-dimensional points

In this part of our experiments, random two-dimensional points were generated. The base of the tolerance relation was the Euclidean distance of these objects ($d$). We defined a similarity ($S$) and a dissimilarity threshold ($D$). $S$ was set to 50 and

Table 3: Average interquartile ranges of the base set sizes for Erdős-Rényi graphs

| **Points** | **HC** | **SHC** | **TABU** | **SA** | **PT** | **GE** | **BEE** | **PSO** | **FF** |
|---|---|---|---|---|---|---|---|---|---|
| **200 q=0.5** | 105 | 92 | 80 | 48 | 45 | 98 | 49 | 92 | 82 |
| **400 q=0.5** | 176 | 189 | 144 | 57 | 78 | 213 | 65 | 186 | 193 |
| **200 q=0.6** | 105 | 94 | 97 | 46 | 46 | 98 | 47 | 89 | 92 |
| **400 q=0.6** | 192 | 173 | 178 | 87 | 62 | 209 | 81 | 193 | 197 |
| **200 q=0.7** | 104 | 94 | 74 | 52 | 64 | 104 | 49 | 104 | 88 |
| **400 q=0.7** | 198 | 179 | 191 | 50 | 53 | 228 | 81 | 186 | 210 |

Table 4: Average numbers of singletons for Erdős-Rényi graphs

| **Points** | **HC** | **SHC** | **TABU** | **SA** | **PT** | **GE** | **BEE** | **PSO** | **FF** |
|---|---|---|---|---|---|---|---|---|---|
| **200 q=0.5** | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 1 |
| **400 q=0.5** | 0 | 0 | 0 | 2 | 2 | 2 | 0 | 2 | 7 |
| **200 q=0.6** | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 2 |
| **400 q=0.6** | 0 | 0 | 0 | 3 | 3 | 1 | 1 | 1 | 3 |
| **200 q=0.7** | 1 | 0 | 0 | 1 | 1 | 1 | 0 | 1 | 2 |
| **400 q=0.7** | 0 | 1 | 0 | 2 | 3 | 1 | 1 | 1 | 4 |

Table 5: Average execution time for Erdős-Rényi graphs

| **Points** | **HC** | **SHC** | **TABU** | **SA** | **PT** | **GE** | **BEE** | **PSO** | **FF** |
|---|---|---|---|---|---|---|---|---|---|
| **200 q=0.5** | 5 | 7 | 163 | 3 | 16 | 82 | 53 | 569 | 274 |
| **400 q=0.5** | 142 | 181 | 1549 | 6 | 39 | 360 | 247 | 7971 | 1167 |
| **200 q=0.6** | 4 | 6 | 170 | 3 | 17 | 91 | 66 | 734 | 317 |
| **400 q=0.6** | 156 | 248 | 1665 | 6 | 40 | 457 | 274 | 8611 | 1287 |
| **200 q=0.7** | 3 | 8 | 156 | 3 | 17 | 87 | 59 | 818 | 283 |
| **400 q=0.7** | 138 | 256 | 1652 | 7 | 43 | 404 | 284 | 9878 | 1336 |

$D$ was set to 90. The tolerance relation $\mathcal{R}$ can be given this way for any objects $A, B$:

$$A\mathcal{R}B = \begin{cases} +1 & d(A,B) \leq S \\ -1 & d(A,B) > D \\ 0 & \text{otherwise} \end{cases}$$

We generated 100, 150, 200, 300, 500 points and each algorithm was run 3 times for each point set and calculated the averages of the values described in section 6. In the following tables, we can see the results.

In Table 6 the average standard deviations of the base set sizes can be seen. In case of a small number of points, the difference was not so considerable, but it became larger as the number of points was increased. In every case, the simulated annealing provided the most acceptable result. It is interesting that the parallel tempering fell short against the simulated annealing for a small number of points. However, in the 500 points test case the difference was negligible. Local search algorithms (hill climbing, its variant, tabu search) were rather good for a small number of points. In almost every situation, the firefly algorithm, genetic algorithm and particle swarm optimization provided the worst result.

Table 7 shows the interquartile ranges of the base set sizes. The outcome was quite similar as in the previous table. For a small number of points, the difference was not so high. For 500 points, it can be more noticeable. Like before, the firefly algorithm, genetic algorithm and particle swarm optimization ended up in the last places, and simulated annealing proved to be the most optimal.

Table 8 shows how many singleton clusters appeared. The results were quite the same in all cases. In is interesting that most of the algorithms were not affected by the increasing number of points. In the 500 points test case, some differences can be observed. In this case, the simulated annealing and its parallel version provided

Table 6: Average standard deviations of the base set sizes

| Points | HC | SHC | TABU | SA | PT | GE | BEE | PSO | FF |
|---|---|---|---|---|---|---|---|---|---|
| **100** | 7.6 | 8.1 | 7.5 | 7.1 | 7.6 | 8.9 | 7.6 | 7.7 | 7.6 |
| **150** | 11.8 | 10.2 | 10.4 | 7 | 11 | 14.5 | 8.7 | 12.2 | 11.5 |
| **200** | 13.1 | 16.2 | 12.8 | 10.3 | 13.2 | 17.8 | 12.7 | 13.8 | 13.3 |
| **300** | 25.7 | 28.9 | 27.2 | 17.1 | 18.4 | 31.4 | 23 | 29.1 | 28.9 |
| **500** | 46.6 | 34.4 | 39.4 | 26.5 | 27 | 51.4 | 34.2 | 47.8 | 48.4 |

Table 7: Average interquartile ranges of the base set sizes

| Points | HC | SHC | TABU | SA | PT | GE | BEE | PSO | FF |
|---|---|---|---|---|---|---|---|---|---|
| **100** | 23 | 24 | 22 | 21 | 26 | 23 | 23 | 26 | 23 |
| **150** | 27 | 25 | 35 | 20 | 31 | 37 | 21 | 37 | 33 |
| **200** | 38 | 38 | 38 | 30 | 41 | 47 | 37 | 40 | 35 |
| **300** | 67 | 68 | 70 | 48 | 61 | 74 | 64 | 71 | 68 |
| **500** | 111 | 94 | 99 | 73 | 81 | 119 | 103 | 117 | 116 |

a rather inadequate result compared to the others. However, this difference was not so high.

In Table 9 the average execution time is listed in seconds. As expected, these values were the most dependent on the number of points. For less than 200 points, the hill climbing algorithm and its stochastic variant provided the fastest run-time. However, after 200 points they could not compete with the simulated annealing which could find the quasi-optimal partition in less than 5 seconds for each test case. The parallel tempering also proved to be quite fast, but not as fast as its single-threaded variant. The other algorithms executed in an unreasonable time which is unacceptable for a great number of points. Especially the particle swarm optimization proved to be very slow, it finished running after 3.5 hours for 500

Table 8: Average numbers of singletons

| Points | HC | SHC | TABU | SA | PT | GE | BEE | PSO | FF |
|---|---|---|---|---|---|---|---|---|---|
| **100** | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| **150** | 1 | 1 | 2 | 0 | 1 | 2 | 0 | 2 | 2 |
| **200** | 1 | 1 | 0 | 0 | 0 | 1 | 0 | 1 | 2 |
| **300** | 0 | 1 | 1 | 2 | 1 | 0 | 0 | 1 | 3 |
| **500** | 2 | 1 | 0 | 5 | 4 | 1 | 3 | 1 | 5 |

Table 9: Average execution time

| **Points** | **HC** | **SHC** | **TABU** | **SA** | **PT** | **GE** | **BEE** | **PSO** | **FF** |
|---|---|---|---|---|---|---|---|---|---|
| **100** | 0.2 | 0.3 | 15.6 | 1.1 | 5.4 | 15.7 | 15.5 | 32.5 | 54.1 |
| **150** | 0.5 | 0.4 | 43.5 | 0.7 | 7.3 | 14.5 | 11.9 | 130.8 | 58.3 |
| **200** | 6.4 | 9.5 | 172.4 | 3 | 16.8 | 92.7 | 66.3 | 564 | 281.1 |
| **300** | 40.1 | 43.5 | 647.4 | 4.7 | 25.2 | 207 | 161.1 | 1937.8 | 579.6 |
| **500** | 69.4 | 108.5 | 3550 | 3 | 50.8 | 207.4 | 135.9 | 13251 | 612.3 |

points.

# 8 Conclusion

In [10] the authors introduced a partial approximation space relying on a similarity relation (a tolerance relation technically). The genuine novelty of approximation spaces is the systems of base sets: it is the result of correlation clustering, and so similarity is taken into consideration generally. Singleton clusters have no real information in approximation process, these clusters cannot be taken as base sets, therefore the approximation spaces are partial in general cases (the unions of base sets are proper subsets of the universes.) The partition, and so the system of base sets, gained from correlation clustering depends on the used search algorithm. In the present paper, we used several algorithms and we showed a way to compare them. In our experiments, we used two different types of random graphs. For these types of graphs, the simulated annealing proved to be best choice. In almost every test case, it provided the most suitable result. However, its most important property is that it was the least affected by the increasing number of points, so it can also finish in reasonable time even for large amounts of points.

# References

[1] Aigner, Martin. Enumeration via ballot numbers. *Discrete Mathematics*, 308(12):2544 – 2563, 2008. DOI: 10.1016/j.disc.2007.06.012.

[2] Aszalós, László and Mária, Bakó. *Advanced Search Methods.* Educatio Társadalmi Szolgáltató Nonprofit Kft., 2012. in Hungarian.

[3] Bansal, Nikhil, Blum, Avrim, and Chawla, Shuchi. Correlation clustering. *Machine Learning*, 56(1-3):89–113, 2004.

[4] Becker, Hila. A survey of correlation clustering. *Advanced Topics in Computational Learning Theory*, pages 1–10, 2005.

[5] Brownlee, Jason. *Clever Algorithms: Nature-Inspired Programming Recipes*. Lulu.com, 1st edition, 2011.

[6] Erdös, P. and Rényi, A. On random graphs i. *Publicationes Mathematicae Debrecen*, 6:290, 1959.

[7] Erdös, P. and Rényi, A. On the evolution of random graphs. In *Publication of the Mathematical Institute of the Hungarian Academy of Sciences*, pages 17–61, 1960.

[8] Mani, A. Choice inclusive general rough semantics. *Information Sciences*, 181(6):1097–1115, 2011.

[9] Mihálydeák, Tamás. Logic on similarity based rough sets. In Nguyen, Hung Son, Ha, Quang-Thuy, Li, Tianrui, and Przybyła-Kasperek, Małgorzata, editors, *Rough Sets*, pages 270–283, Cham, 2018. Springer International Publishing.

[10] Nagy, Dávid, Mihálydeák, Tamás, and Aszalós, László. 10.1007/978-3-319-60840-2_7, *Similarity Based Rough Sets*, pages 94–107. Springer International Publishing, Cham, 2017.

[11] Nagy, Dávid, Mihálydeák, Tamás, and Aszalós, László. Similarity based rough sets with annotation. In Nguyen, Hung Son, Ha, Quang-Thuy, Li, Tianrui, and Przybyła-Kasperek, Małgorzata, editors, *Rough Sets*, pages 88–100, Cham, 2018. Springer International Publishing.

[12] Pawlak, Zdzisław. Rough sets. *International Journal of Parallel Programming*, 11(5):341–356, 1982.

[13] Pawlak, Zdzislaw et al. Rough sets: Theoretical aspects of reasoning about data. *Systern Theory, Knowledge Engineering and Problem Solving, Kluwer Academic Publishers, Dordrecht, 199l*, 9, 1991.

[14] Pawlak, Zdzisław and Skowron, Andrzej. Rudiments of rough sets. *Information sciences*, 177(1):3–27, 2007.

[15] Skowron, Andrzej and Stepaniuk, Jaroslaw. Tolerance approximation spaces. *Fundamenta Informaticae*, 27(2):245–253, 1996.

[16] Yao, Yiyu and Yao, Bingxue. Covering based rough set approximations. *Information Sciences*, 200:91 – 107, 2012. DOI: `http://dx.doi.org/10.1016/j.ins.2012.02.065`.

[17] Zimek, Arthur. Correlation clustering. *ACM SIGKDD Explorations Newsletter*, 11(1):53–54, 2009.

Acta Cybernetica **24** (2019) 121–129.

# LZ based Compression Benchmark on PE Files

Zsombor Paróczi[a]

**Abstract**

The key element in runtime compression is the compression algorithm itself, that is used during processing. It has to be small in enough in decompression bytecode size to fit in the final executable, yet have to provide the best possible compression ratio. In our work we benchmark the top LZ based compression methods on Windows PE (both EXE and DLL) files, and present the results including the decompression overhead and the compression rates.

**Keywords:** lz based compression, compression benchmark, PE benchmark

## 1 Introduction

During runtime executable compression an already compiled executable is modified in ways, that it still retains the ability to execute, yet the transformation produces smaller file size. The transformations usually exists from multiple steps, changing the structure of the executable by removing unused bytes, adding a compression layer or modifying the code itself. During the code modifications the actual bytecode can change, or remain the same depending on the modification.

In the world of x86 (or even x86-64) PE compression there are only a few benchmarks, since the ever growing storage capacity makes this field less important. Yet in new fields, like IOT and wearable electronics every application uses some kind of compression, Android apk-s are always compressed by a simple gzip compression. There are two mayor benchmarks for PE compression available today, the Maximum Compression benchmark collection [1] includes two PE files, one DLL and one EXE, and the PE Compression Test [2] has four EXE files. We will use the 5 EXE files PE files during our benchmark, referred as *small corpus*. For more detailed results we have a self-collected corpus of 200 PE files, referred to as *large corpus*.

When approaching a new way to create executable compression, one should consider three main factors. The first is the actual compression rate of the algorithms, since it will have the biggest effect on larger files. The second is the overhead in terms of extra bytecode within the executable, since the decompression algorithm have to be included in the newly generated file, using large pre-generated dictionary is usually not an option. This is especially important for small (less than

[a]Budapest University of Technology and Economics, E-mail: paroczi@tmit.bme.hu

DOI: 10.14232/actacyb.24.1.2019.9

100kb) executables. The third factor has the lowest priority, but still important: the decompression speed. The decompression method should not require a lot of time to run, even on a resource limited machine. This eliminates whole families of compression methods, like neural network based (PAQ family) compressions.

```
83 e6 01          and   esi, 1
8d 3c 96          lea   edi, dword ptr [esi+edx*4]
39 44 b9 1c       cmp   dword ptr [ecx+edi*4+1ch], eax
75 77             jne   short L15176
Opcode+ModRM — Jump Offset — Displacement — SIB — Immediate
```

Figure 1: Annotated asm code

Split-stream methods are well-know in the executable compression world, these algorithms take advantage of the structural information of the bytecode itself, separating the opcode from all the modification flags. Each x86 instruction can be separated into multiple parts, prefix, opcode, mod r/m, etc., an annotated asm snippet can be seen on Figure 1. The idea behind split-stream is to annotate each byte by these parts, and collect them into one chunk. By doing this, each chunk can be compressed better due to local redundancies. During decompression the original bytecode is reconstructed using a small compiler. We used a reference implementation from the packer kkrunchy [6].

## 2 LZ based compression methods

LZ based compression methods (LZ77/LZSS/LZMA families) are well fitted for this compression task, since they usually have relatively small memory requirement (less than 64 Mb), they use Lempel-Ziv compression methods [3] and maybe some Huffman tables or hidden Markov model based approaches. These methods are simple algorithms, resulting in small size in terms of decompression bytecode. During the last few years there are a lot of new LZ based compression methods, the mayor ones are Zstandard (zstd) from Facebook and Zopfli from Google. The selected libraries can be seen on Table 1, these are the top LZ familly libraries for generic purpose compression regarding an extensive LZ benchmark [4,5].

The compression rates on generic dataset (non-code section of an executable) can be seen on Figure 2 and Table 2. All of these tests and results are in sync with the LZ benchmark mentioned previously, the only exception is Brotli which worked quite well on our dataset. Brotli, Lzlib and LZMA have the best compression ratio on average, followed by CSC, Zopfli and zstd. aPlib has the worst compression ratio, since it only implements a very simple LZ77 variant.

## 3 Decompression code

For each compression method the related library also supplies the decompression method as well. In most cases it's tightly coupled with the compression code, so

Table 1: Libraries used in the benchmark

| Compression method | Version | Source |
|---|---|---|
| aPlib | 1.1.1 | http://ibsensoftware.com/products_aPLib.html |
| Lzlib | 1.10 | https://www.nongnu.org/lzip/lzlib.html |
| LZMA | 9.35 | https://www.7-zip.org/sdk.html |
| Zopfli | 2017-07-07 | https://github.com/google/zopfli |
| Zstandard | 1.3.3 | https://facebook.github.io/zstd/ |
| CSC | 2016-10-13 | https://github.com/fusiyuan2010/CSC |
| Brotli | 1.0.3 | https://github.com/google/brotli |

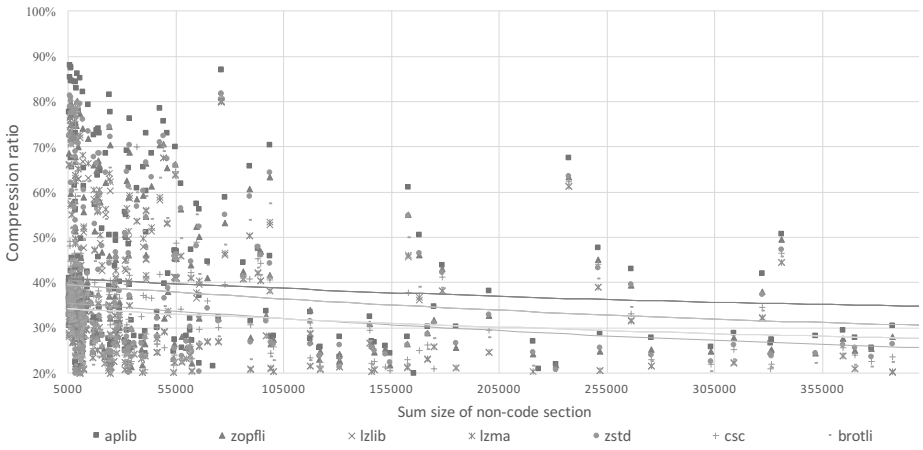



Figure 2: Compression rates on non-code section by input size

Table 2: Compression rates on non-code section

| Method | aPlib | Zopfli | Lzlib | LZMA | zstd | CSC | Brotli |
|---|---|---|---|---|---|---|---|
| Rate | 40.2% | 37.2% | 34.7% | 34.2% | 38.3% | 34.6% | 33.4% |

in the first step we separated the compression method from the decompression one and created small executables which included only the decompression method and a sample from the compressed data, so we can verify that the decompression still works. All these LZ based compression libraries are written in C / C++, during this step we used GCC for ease of debugging. The aPlib library includes an ASM written decompression method which is already small enough, so we didn't do any modification on it. Lzlib, LZMA, zlib, zstd, CSC and Brotli are at some point use

dynamic memory allocation, Brotli and CSC has some other external dependencies. We opted to remove (or inline) all dependencies, since loading external DLLs or extra functions takes up more space than an inlined function. We managed to fully remove memory allocation from LZMA and Lzlib by simply creating a large chunk of zerofilled memory at the end of the executable, and absolutely referencing those with some pointers. In the other libraries we could inline some trivial functions (zerofill, memcpy) but due to the nature of those algorithms there are a lot of dynamic allocations that require external libraries. We also remove all error reporting functionality from the code, these are designed to detect if the compressed data is damaged in any way. All of the modifications were tested with multiple samples to retain the ability of decompressing compressed data.

Table 3: Decompression bytecode size

| Compression method | Bytecode size | Compressed with aPlib |
|---|---|---|
| aPlib | 150 | - |
| Lzlib | 7.168 | 3.943 |
| LZMA | 8.602 | 3.155 |
| Zopfli | 14.351 | 8.173 |
| Zstandard | 106.525 | 26.632 |
| CSC | 23.714 | 10.671 |
| Brotli | 215.665 | 92.736 |

GCC has several flags for optimizing for space, speed and even some internal optimization options are available, but after several failed attempts to make the pure decompression bytecode smaller, we started to experiment with other compilers. Clang and Microsoft Visual C++ compiler produced almost the same bytecode size, even with extra optimization options, but Watcom Compiler (Open Watcom 1.9) managed to create 10%-15% smaller bytecode than any of the other compilers (second best was gcc with size optimization flags). This is due to the fact that generic registers (registers storing and passing variables between functions) can be fine-tuned in Watcom, using esi, edi, ebp registers in the produced binaries. After several iterations of modifying the code, testing and compiling we managed to create really small sized decompression code for each library. We also noticed that compressing the various decompressing bytecodes with aPlib and decompressing them during runtime is a great way to create smaller sized binaries. The aPlib decompression bytecode is 150 bytes after all. Table 3 contains the bytecode size on both the decompression code bytesize as is, and the compressed decompression bytecode size. Also worth noting that Brotli can be compiled without the built-in dictionary, which results in 66.930 bytes (and 23.425 bytes compressed with aPlib), but the dictionary has huge benefits during compression / decompression. Any data compressed with Brotli with dictionary can only be decompressed, if the decompressor code also has the dictionary.

# 4 Benchmark

During the benchmark we constructed a system, which is capable of extracting different sections from the executables, apply split-stream and a compression method on it to create a well detailed benchmark result. During the benchmark we run each compression method on each section, then run each compression method with split-stream on executable sections. The benchmark system was created using C++ and Node.js, the Node.js part was responsible for the instrumentation of the compressions, the C++ part was responsible for extracting the section and verifying the modified decompression method we created. If there is any side effect from the decompression code modification explained in the third section, we are not seeing it.

# 5 Results: compression ratio

The detailed results for each test case on the *small corpus* can be seen on Figure 3. As you can see applying split-stream before the compression is useful in most of the cases (except for the smallest executable, which suffered from the overhead of this method - splitting 1 byte instructions into base instruction + mod flags). The rates for each compression varies between test cases, but Lzlib, LZMA, Brotli

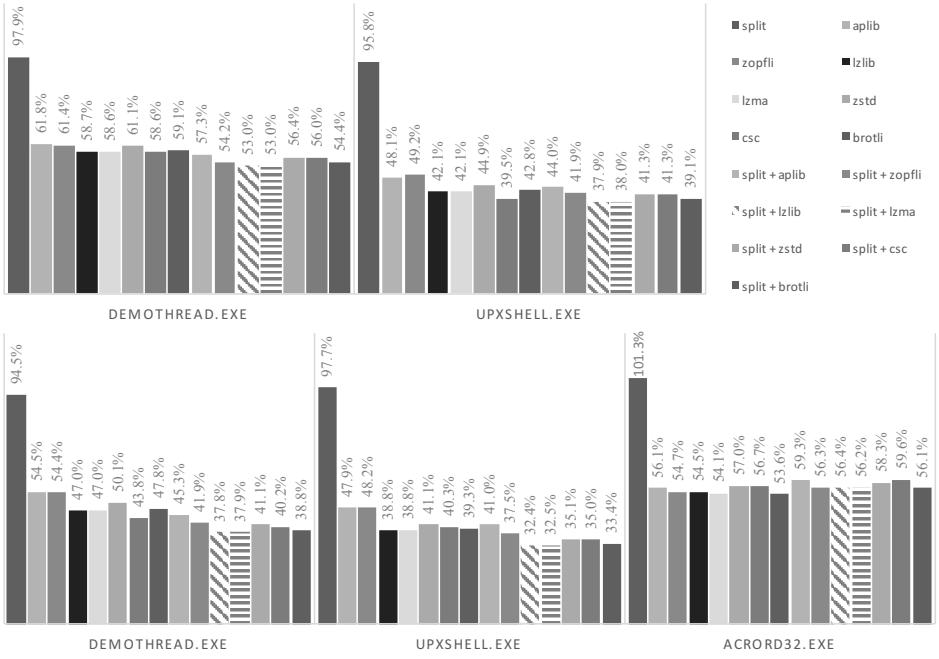



Figure 3: Resulting section size compered to the original on the *small corpus* files

are clearly the best for the *small corpus*, followed by zstd, CSC, Zopfli and aPlib. There is a constant improvement when using split-stream. Only for really small executable aPlib is the best, due to the simplicity of the algorithm itself. All of these results were verified during our *large corpus* benchmark.

The actual compression rates on the *large corpus* can be seen on Table 2 and 4 (split-stream is annotated as s). As you can see the ratio between each compression rate on average is really small, for code sections split-stream really helps. For code section LZMA, Lzlib and Brotli are the best, followed by Zopfli and CSC. For non-code section we had a larger variety of results, since the non-code sections can contain any datatype. The non-code section has a more loose structure and less density, the compression rates are higher. It is interesting, that Brotli is the winner in these tests, but as it turned out Brotli has a large dictionary prebuilt into the

Table 4: Average compression rates on code section

| Compression method | compression rate |
|---|---|
| aPlib | 47.0% |
| LZMA | 42.1% |
| s + aPlib | 44.3% |
| s + Zopfli | 41.3% |
| s + Lzlib | 39.6% |
| s + LZMA | 39.5% |
| s + zstd | 42.4% |
| s + CSC | 42.0% |
| s + Brotli | 40.0% |

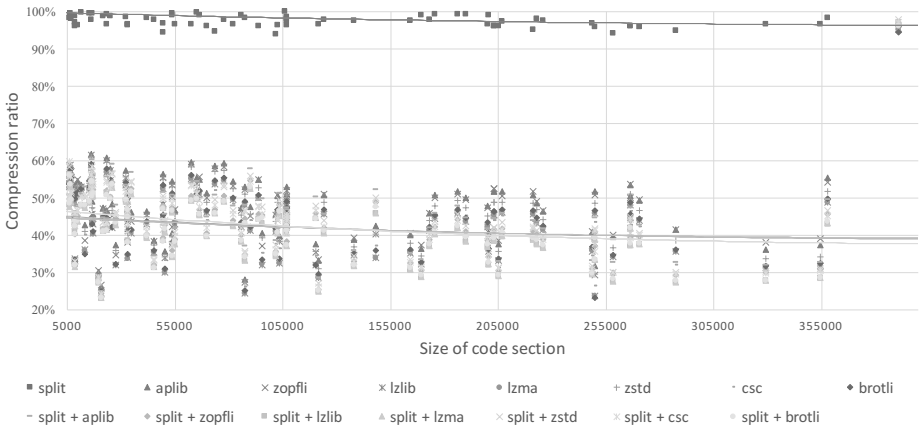



Figure 4: Compression rates vs file size the code section

algorithm, that helps with compressing text. LZMA, Lzlib, CSC produced just 1-2% lower rates, followed by zstd and Zopfli. Obviously aPlib was the worst in both tests, since it contains the most simple algorithm for compression. PE sections tend to be less then 3 Mb, the larger the section the more compression rate we can achieve.

# 6 Results: final file size with decompression bytecode

Since the decompression code has to be included in the final executable, we also benchmarked how the decompression overhead code effects the final file size. As you can see on Figure 4 for smaller executables the overhead is what really defines the final result. All of the decompression methods were packed with aPlib, since aPlib has a decompression code size of 150 bytes, and above 1.000 bytes it is better to compress the decompression code with aPlib. Some of the more complex methods (namely zstd, Brotli, CSC) has relatively large data tables in the decompression code. Same goes for the split-stream code, which is above 1kByte uncompressed, and 540 byte compressed with aPlib.

Our final results suggest, that there is no "golden" LZ based compression with split-stream method for all the executables.

You can see the best performing algorithm on Figure 6 for the *large corpus*. For smaller files a more detailed result of this can be seen on Figure 5. There is a clear

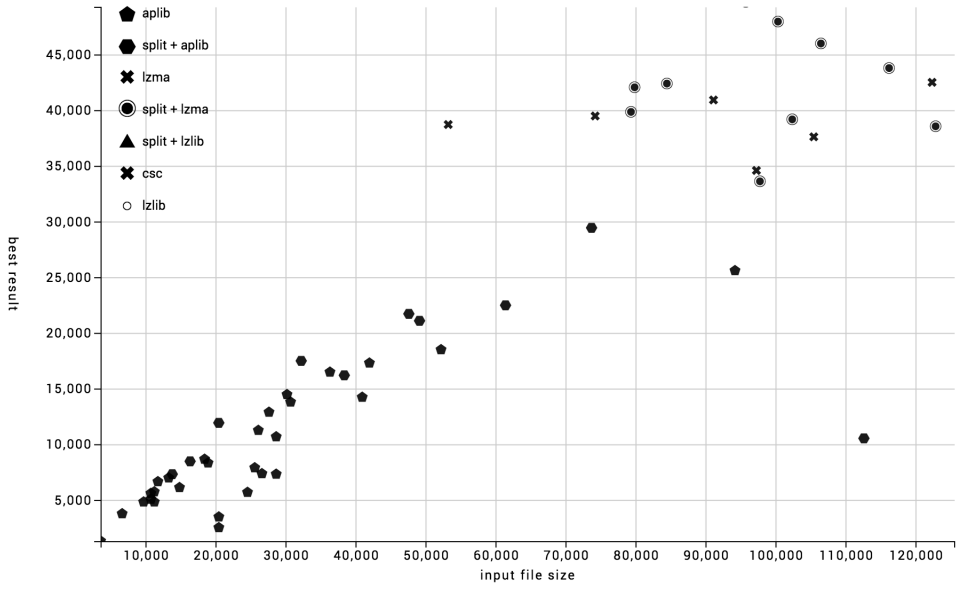



Figure 5: Raw and compressed file size using the best method on smaller files

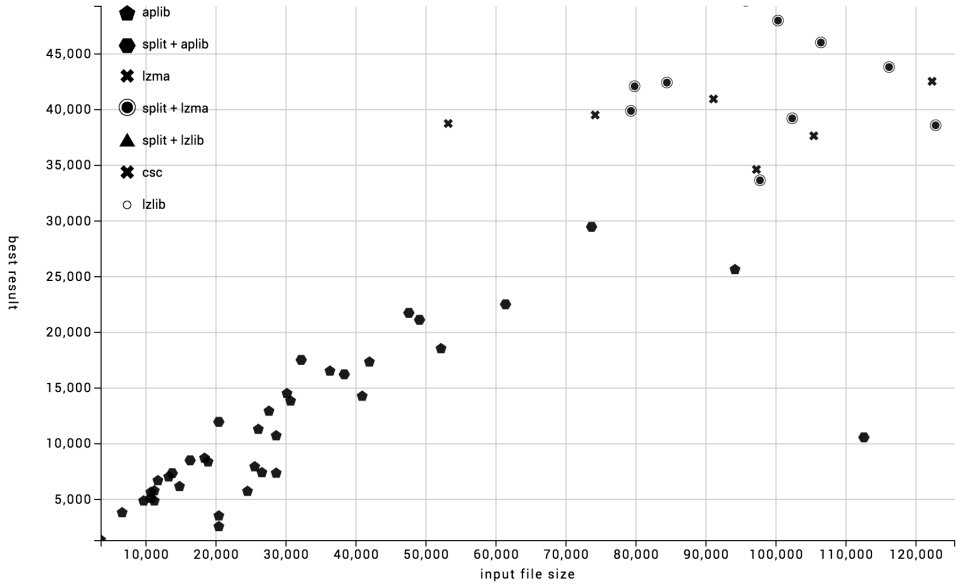



Figure 6: Raw and compressed file size using the best method on larger files

tendency, that some algorithms perform better on smaller files, partly due to the fact that the decompression code is small, and others perform well on larger files. Since smaller files tend to be more code section heavy, and larger files are more like a generic datafile (much more strings, xml, images within the sections), there is an interesting trend how each compression method behaves on different sized binaries.

We consider 3 categories based on the executable size: for small files (less than 50kB) size aPlib is the clear winner with 150 byte decompression code, maybe with split-stream if the executable section is large. For medium size (less than 500 kB) split-stream with aPlib or split-stream with LZMA (aPlib compressed) should be used. For larger files split-stream with LZMA (aPlib compressed) or split-stream with Lzlib (aPlib compressed) should be used.

For some special cases any combination can be the winner in the final compression size. CSC (without split-stream), Lzlib (without split-stream) and LZMA (without split-stream) can outperform the others in some cases.

# 7 Summary

By providing a good ruleset for choosing the right compression method or methods based on file size, we hope that future executable compression authors can improve the compression rate of their tools. Besides that we see a clear trend, that even LZ based compression libraries are getting more complex (dynamic memory allocations, large dictionary size, etc.), making small size, compact decompression bytecode

creation a lot harder. We provided our insights of how to make small decompression bytecode by simply modifying the decompression method in different ways, using different compilers and compressing the bytecode itself.

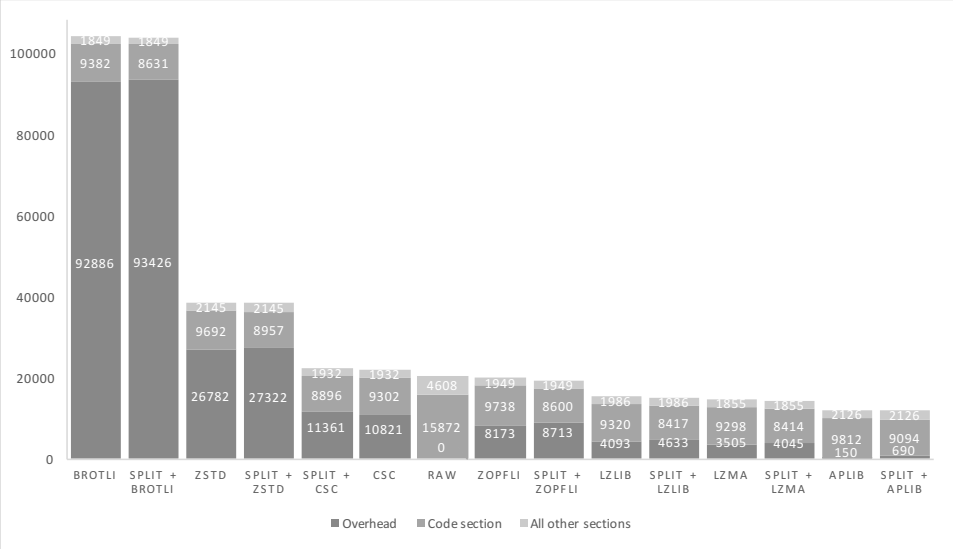



Figure 7: Final executable size example on DemoThread.exe (20kByte)

# References

[1] Lossless data compression software benchmarks/comparisons. `https://www.maximumcompression.com/` (Visited 2018-03-04).

[2] PE Compression test by Ernani Weber. `http://pect.atspace.com/` (Visited 2018-03-04).

[3] Ziv, J. and Lempel, A. A universal algorithm for sequential data compression. *IEEE Trans. on Inf. Th. IT-23*, 337-343, 1977.

[4] LZbench. `https://github.com/inikep/lzbench/` (Visited 2018-03-04).

[5] Kunkel, Julian. SFS: A Tool for Large Scale Analysis of Compression Characteristics *Research Papers (4)*, Research Group: Scientific Computing, University of Hamburg

[6] Giesen, Fabian. Working with compression. *Breakpoint conference*, 2006.

Acta Cybernetica **24** (2019) 131–155.

# A Preparation Guide for Java Call Graph Comparison: Finding a Match for Your Methods*

Zoltán Ságodi[a] and Edit Pengő[a]

**Abstract**

Call graphs provide a basis for numerous interprocedural analyzers and tools, therefore it is crucial how precisely they are constructed. Developers need to know the features of a call graph builder before applying it to subsequent algorithms. The characteristics of call graph builders are best understood by comparing the generated call graphs themselves. The comparison can be done by matching the corresponding nodes in each graph and then analyzing the found methods and calls.

In this paper, we developed a process for pairing the nodes of multiple call graphs produced for the same source code. As the six static analyzers that we collected for call graph building handles Java language elements differently, it was necessary to refine the basic name-wise pairing mechanism in several steps. Two language elements, the anonymous and generic methods, needed extra consideration. We describe the steps of improvement and our final solution to achieve the best possible pairing we are able to provide, through the analysis of the Apache Commons-Math project.

**Keywords:** call graph, Java, static analysis

## 1 Introduction

Static source code analyzers play an important role in producing high-quality software that satisfies the requirements of today's industrial development. They help programmers eliminate flaws and rule violations early on by automatically analyzing the subject system and highlighting its potentially erroneous parts. Usually, the source code is converted into an Abstract Syntax Tree[1] (AST) - like representation,

*This research was supported by the EU-funded Hungarian national grant GINOP-2.3.2-15-2016-00037 titled "Internet of Living Things". This research was supported by the project "Integrated program for training new generation of scientists in the fields of computer science", no EFOP-3.6.3-VEKOP-16-2017-0002. The project has been supported by the European Union and co-funded by the European Social Fund.

[a]University of Szeged, Department of Software Engineering, E-mail: {sagodiz,pengoe}@inf.u-szeged.hu

[1]Abstract Syntax Tree represents the syntactic structure of the source code in a hierarchical tree-like form

DOI: 10.14232/actacyb.24.1.2019.10

which is the basis for further transformations, optimizations, and operations, for example, call graph creation. The capabilities of such analyzer tools depend on the complexity of the internal representations and algorithms they use.

Call graphs are directed graphs representing control flow relationships among the methods of a program. The nodes of the graph denote the methods, while an edge from node $a$ to node $b$ indicates that method $a$ invokes method $b$. Call graphs are essential building blocks of interprocedural control and data flow modeling. They can be used during control flow analysis, program slicing, program comprehension, bug prediction, refactoring, bug-finding, verification, security analysis, and whole-program optimization [8, 13, 35, 37]. The accuracy of call graphs influences the results of the subsequent analyses, consequently, careful consideration is needed during the selection of the construction method. The most obvious difficulty that a static call graph builder has to face is the handling of polymorphic calls and other cases when the target of a call depends on the runtime behavior of the program. There are plenty of call graph builder algorithms that address this challenge and try to make assumptions about what methods could be called. They have extensive literature, including detailed comparisons [14, 15, 20, 21, 24, 32]. There are other factors that can cause differences in the output of two call graph creator tools, for example, the handling of different kinds of initializations or anonymous classes.

In the future, we plan to compare and characterize call graph builders based on how they handle such factors. However, the first step towards this goal is to make the produced call graphs comparable. To compare the call graphs that were generated by different tools for the same source code, we have to match the nodes – i.e. the methods – that correspond to each other and then evaluate what types of methods and calls were found by each tool. However, matching the methods to each other is challenging, since it is not certain that all tools will find the same methods or they might name them differently. Using the line information for refining the pairing mechanism can also cause difficulties. Although node pairing is the basis of call graph comparison, we found no satisfactory description about it in previous works. Therefore, we decided to summarize the problems we encountered and our attempts at solving them. Section 2 provides the related work, whilst Section 3 introduces the investigated call graph builders. Section 4 illustrates the obstacles of the method pairing mechanism and our step-by-step improvements with results. In Section 5, our approach is compared with a topology based solution. Finally, we draw our conclusions and outline future work in Section 6.

## 2 Related work

The way to compare the capabilities of call graph builder tools is through comparing the call graphs they generate. Due to the increasing number of extremely large graphs and their wide area of usability, there are many algorithms and metrics available for comparing general directed and undirected graphs [19, 22, 34]. However, these methods cannot be directly applied to call graphs, especially if they were produced by different analyzer tools. Call graphs are directed graphs whose

nodes correspond to the methods in the source code. Even if the structure of two call graphs is isomorphic, they can be considered entirely different because of the labeling of the nodes. Therefore, to make them comparable, first we have to find a mapping, which is the aim of this paper.

There are several proposals for comparing labeled graphs if a mapping is already present. Champin and Solnon defined a similarity with respect to a given mapping between two graphs that have multiple labels both on their nodes and edges [7]. A graph is described by the set of all of its features, e.g. the set of node-label and edge-label pairs. The similarity measure is calculated based on a simplified version of Tversky's formula [33]. They also proposed an algorithm for finding the best mapping for reaching the maximum similarity, which provides a qualitative description of the differences between the two graphs.

There are algorithms available exactly for comparing labeled graphs that share the same node set [38]. In case of call graphs that were produced by different tools and algorithms this condition cannot be ensured. The simplest way to compare graphs with the same node set is to handle the adjacency matrices as vectors and calculate an edit distance, in other words, the number of different edges [12, 30]. Wicker et al. introduced a dissimilarity measure for graphs like these based on their eigenvalues and eigenvectors [38], which takes into account the global graph structures as well.

The precision and structure of the call graphs greatly depends on the algorithms that the builder tools used. If several call targets are possible for a given call site, more examination is needed to determine which edges should be connected. There are context-dependent and context-independent solutions; naturally, the choice influences the result. Context-dependent methods are more accurate, but in return they use more resources. To mitigate the resource demands of such methods, the analysis of the programs often starts only from the `main` method or a few entry points instead of starting from every method of the analyzed source code. This, however, will likely lower the accuracy of the method. Context-independent methods for object oriented languages can be improved with the following algorithms: *CHA* [9], *RTA* [4], *XTA*[32], *VTA* [31]. In case of the comparing these call graph building strategies [1, 14, 15], node matching is usually not an issue because the algorithms are implemented in the same environment and language elements are handled similarly. The nodes of the produced call graphs are the subset of each other's node set with the same naming convention, therefore, the main difference comes from the number of edges.

In this paper, we considered the results of static analyzer tools, meaning that we worked with the so called static call graphs. However, call graphs can be composed with dynamic tools as well from actual executions of the analyzed program. Lhoták [20] compared static call graphs generated by Soot [29] and dynamic call graphs created with the help of the *J [28] dynamic analyzer. He built a framework to compare call graphs, discussed the challenges of the comparisons, and presented an algorithm to find the causes of the potential differences in call graphs. The paper does not describe the difficulties of matching the nodes of the call graphs that were provided by different sources. The reason could be that Soot is a bytecode

analyzer, therefore its output is close to the output of a dynamic analyzer. This also means that our work could be easily extended by including such dynamic call graphs.

Murphy et al. [24] carried out a study about the comparison of five static call graph creators for C in 1996. The outputs of the analyzers were compared to a baseline call graph created by the GCT test coverage tool, which was based on the GNU C compiler. They identified significant differences in how the tools handled typical C constructs, like macros. Mapping the graphs was made more complicated depending on which files were involved in the analysis. They applied a filtering mechanism to solve this. Other difficulties of the matching mechanism were not discussed, although C functions are clearly identified by their name only.

Naturally, it is possible to compare two graphs by considering only the structure of the graph without label information. Many similarity measures are based on iterative calculations. These methods repeatedly refine an estimated initial similarity value of the graph nodes by using an update formula. The update formulas consider the similarity of the edges and the neighboring nodes. When a termination condition is met the iteration finishes and a similarity matrix is produced. Nikolič proposed an iterative solution [25] called neighbor matching that addresses the insufficiencies of the previously existing approaches [5, 23, 39, 16]. An in - and out - similarity is defined for the update formula. To determine the in-similarity an optimal matching of in-neighbors has to be constructed. The calculation is analogous in case of the out-similarity. The introduced node similarity calculation was evaluated on isomorphic subgraph matching and on a social network, and concluded that it is more accurate than the previous approaches. Nikolič provided a C++ implementation of the proposed method. In Section 5 we compare the results of this topology based graph similarity tool with our pairing mechanism.

## 3 Analyzed tools

We studied numerous static analyzer tools for Java to decide whether they could generate – or could be easily modified to generate – call graphs. We aimed for widely available, open-source programs from recent years, which could analyze complex, real-life Java systems. The diversity of the tools was another important aspect of our selection criteria. We involved tools that provide a direct interface for call graph creation, whilst, in other cases, the graph had to be extracted directly from the inner representation of the analyzer. The investigated analyzers can also be categorized by whether they work on source or byte code, which, of course, affects their results. Most of the tools are command line based, although an Eclipse plug-in based solution was also examined. The selected analyzers support several call graph-creation algorithms, which greatly influences the characteristics, the accuracy and the size of the generated graph. It is the application that determines what type of call graph is the most useful, sometimes a small and less accurate call graph is better, while, in other cases, a large and precise one is needed. The goal of this paper is not to compare the output of these call graph-builder algorithms, but to

pair the corresponding graph sections, which is the basis for further comparative studies. Therefore, we considered the algorithm only as an attribute of the given tool.

Table 1 summarizes the most important properties of the examined call graph builder tools. The grey lines correspond to two of the discarded tools that we tested in more detail. In both cases, the reason for the exclusion was their lack of robustness. For example, JavaParser [18] did not give enough information to reconstruct the caller-callee relationships between compilation units without major development. Call Hierarchy Printer[6] (CHP) failed to finish the analysis of projects larger than a few thousands lines of code. The selected tools, the analyzed sources, and the results are available as an online appendix[2].

The description of the six tools that were selected for the comparison is presented below.

Table 1: Summary of examined call graph creator tools

| | **version** | **maintained** | **input** | **robustness** | **built-in call-graph construction** | **multiple algorithms available** |
|---|---|---|---|---|---|---|
| **JCG** | commit da81eeb on Oct 24 2018 | ✓ | byte code | ✓ | ✓ | × |
| **SPOON** | 7.0.0 | ✓ | source code | ✓ | × | × |
| **WALA** | 1.5.1 | ✓ | byte code | ✓ | ✓ | ✓ |
| **OSA** | 1.0.0 | ✓ | source code | ✓ | × | × |
| **Soot** | 3.2.0 | ✓ | byte code | ✓ | ✓ | ✓ |
| **JDT** | Eclipse Oxygen.2 (4.7.2) | ✓ | source code | ✓ | × | × |
| **CHP** | commit 3316b4a on Mar 26 2015 | × | both | × | × | × |
| **JavaParser** | 3.5.16 | ✓ | source code | × | × | × |

## 3.1 Java Call Graph

The Java Call Graph (JCG) [17] is an Apache BCEL [3] based utility for constructing static and dynamic call graphs. It can be considered a small project as it only has one major contributor, Georgios Gousios, whose last commit (at the time of this writing) is from October, 2018. It supports the analysis of Java 8 features and requires a `jar` file as an input. A special feature of the analyzer is the detection of *unreachable*[3] code. As a result, the call graph does not include calls from code segments that are never executed.

## 3.2 SPOON

SPOON [27] is an open-source, feature-rich Java analyzer and transformation tool for research and industrial purposes. It is actively maintained, supports Java up

[2] `http://www.inf.u-szeged.hu/~pengoe/research/StaticJavaCallGraphs/`

[3] Unreachable code will never be executed as there is no control flow path to it from the entry point of the program.

to version 9, and while several higher-level concepts (e.g., reachability) are not provided "out of the box", the necessary infrastructure is accessible for users to develop their own. SPOON performs a directory analysis[4] of the source code and builds an AST-like metamodel, which is the basis for these further analyses and transformations. We extracted the call graph of our project by traversing this internal representation and collecting every available invocation information. The library is well-documented and provides a visual representation of its metamodel, which helped us in thoroughly studying its structure.

## 3.3 WALA

WALA [36] is a static and dynamic analyzer for Java bytecode (supporting syntactic elements up to Java 8) and JavaScript. Originally, it was developed by the IBM T.J. Watson's Research Center; now it is actively developed as an open-source project. WALA has a built-in call graph generation feature with a wide range of graph building algorithms. We used the *ZeroOneContainerCFA* graph builder for our experiments, as it performs the most complex analysis. It provides an approximation of the Andersen-style pointer analysis [2] with unlimited object-sensitivity for collection objects. The generator has to be parameterized with the entry points from which the call graphs would be built. To make the results similar to the results of the other tools, we treated all the methods as entry points (instead of just the `main` methods). For other configuration options, we used the default settings provided in the documentation and example source codes.

## 3.4 OpenStaticAnalyzer

OpenStaticAnalyzer (OSA) [26] is an actively maintained, multi-language static analyzer framework developed by the Department of Software Engineering at the University of Szeged. It calculates source code metrics, detects code clones, performs reachability analysis, and finds coding rule violations up to Java 8. Other languages such as Python and C# are supported as well. Besides the directory analysis of the source code, OSA is also capable of wrapping the build system (*maven* or *ant*) of the project under examination. This can make the analysis more precise as generated files will be handled too. Similarly to the above mentioned SPOON implementation, we extracted the call graphs by processing the AST-like inner representation of OSA.

## 3.5 Soot

Soot [29] is a widely used language manipulation and optimization framework developed by the Sable Research Group at the McGill University. It supports analysis up to Java 9 and works on the compiled binaries. Although its official website[5] has

[4] The static analyser processes recursively every Java file in a given root folder

[5] `https://www.sable.mcgill.ca/soot/soot_download.html`

the latest release from 2012, the project is active on GitHub, from where we acquired the 3.2.0 release, which was the latest version then. Like WALA, Soot also has a built-in call graph creator functionality. For the analysis of library projects the CHA algorithm was used for call-graph construction, while in case of standalone projects we used the SPARK framework, which employs a points-to analysis algorithm.

### 3.6 Eclipse JDT

The Eclipse Java development tools (JDT) [10] is one of the main components of the Eclipse SDK [11]. It provides a built-in Java compiler and a full model for Java sources. We created a JDT based plugin for Eclipse Oxygen that supports even Java 10 code, to extract the call graph from the extensive, AST-like inner representation.

## 4 Refining the pairing mechanism

There are numerous elements that could cause differences in call graphs, as tools process language elements differently. In this section, we discuss what attempts we made to handle these differences and what were the benefits and downsides to each approach. In this article, the pairing mechanism is illustrated through the Apache Commons Math 3.6.1[6] project (208,876 KLOC). We are only using one project as an example, since our aim is to showcase the process itself, not to compare data. More analyzed projects are presented in the online appendix mentioned in Section 3.

### 4.1 Overview of process

The following four subsections correspond to the process of developing a unified representation for Java method names. Figure 1 provides an overview of this development process. It was previously stated that the call graph creator tools produce the graphs in slightly different formats. Therefore, we had to implement a specific graph loader for each tool to handle the aspects of its method naming convention. A basic name pairing (1.) was introduced to treat the fundamental differences of the representations. However, anonymous language elements needed extra consideration for which the anonymous transformation method (2.) was developed. As the figure indicates this heuristical approach is not part of the final approach. We found that the anonymous transformation method could be improved by using line information (3.). This introduced a challenge in the handling of generic source code elements which had to be dealt with (4.). No other Java language elements were identified that impaired the pairing mechanism.

[6] `http://commons.apache.org/proper/commons-math/`

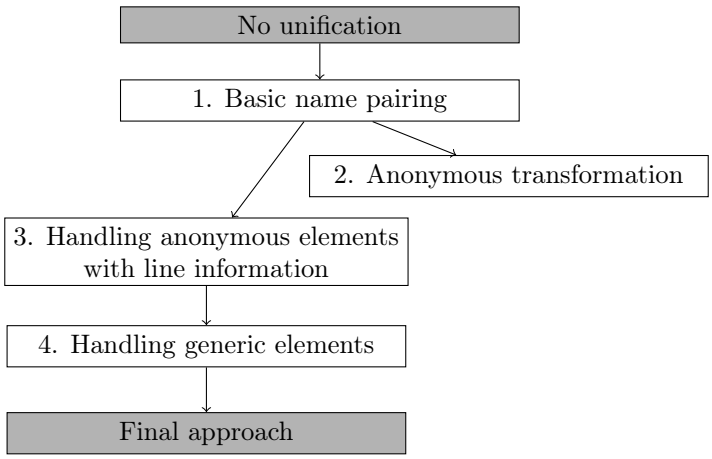



Figure 1: Development process

## 4.2 Basic name pairing

In Java, methods can be distinguished by fully qualified names, which include the package name, the class name, the name of the method, and the list of the parameter types. The return value is not required for the identification, however, we encountered one case where it is indeed needed. According to the Java standard, overridden methods can differ in return type if the return-type-substitutability is satisfied, for example, the child class specializes the return type to a subtype. Some tools represent both the specialized and the not-specialized methods for a child class, although they only connect edges to one of them. Therefore, these rare cases could be easily detected.

Call graph comparison is based on identifying and matching the corresponding methods in each graph regardless of their representation. At first, we only used the method names produced by the static analyzers as basis of the method pairing. However, this was not enough because some fundamental features are represented differently, for example, some of the tools denote constructor methods with the class name, whilst others tag them with the name `<init>`. Therefore, we developed a common representation for the Java methods and as a first step of the comparison we transformed every call graph to this unified representation. Only the constructor methods, initializer blocks and other not-so-significant representational differences were subject to the name unification process. Instance initializer blocks are executed every time an instance of that class is created. They can be used to initialize class members. The Java compiler copies initializer blocks into every constructor. Therefore, initializer blocks can be used to share a block of code between multiple constructors. Byte code analyzer tools, such as WALA, represent initializer blocks as part of the constructor methods. However, source code analyzers such as SPOON represent the initialization blocks and the constructor methods with

separate nodes for the given class. Our basic name pairing method aggregates the nodes of initializers blocks with every constructor of that class, making it pairable with the constructor methods found in the compared graph. This functionality can be turned off with a command line option, if the user wishes.

Figure 2 - 4 help in understanding the process of basic name pairing. Figure 2 shows a sample code containing constructors and initializer blocks. The reason we only included these two language elements in the sample code is because during basic name pairing only they require special consideration. The call graphs of the sample code are portrayed in Figure 3. These are produced by two of the tools, SPOON and WALA. The grey nodes belong to SPOON's graphs, the white ones belong to WALA's graph. As described in the previous paragraph, SPOON represents initializer blocks with separate nodes, while WALA treats them as part of the constructor methods, which causes a slight discrepancy between the two graphs. For this reason, during our pairing mechanism we aggregate the nodes of the initializer blocks with constructor nodes to ensure that none of them remain without a pair. A method name unification is also performed on each of the nodes. The results of the aggregation and the unification process can be seen in Figure 4. The borders of the rectangles indicate which nodes are paired between the two graphs. Two nodes are paired if their names match character by character.

```
class Test {
  Integer i;
  public Test(){}
  public Test(String s){}
  {
    i = new Integer(89);
  }
}
```

Figure 2: The basic name pairing in action: sample code

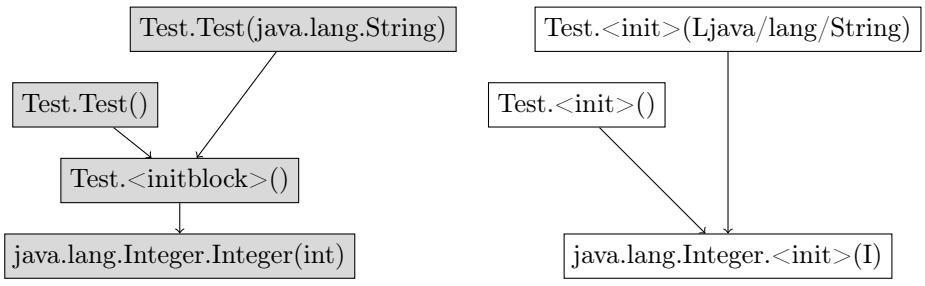



Figure 3: The basic name pairing in action: input graphs

Table 2 summarizes the results of this initial attempt on the Commons Math project. The diagonal elements in bold show the number of different methods found

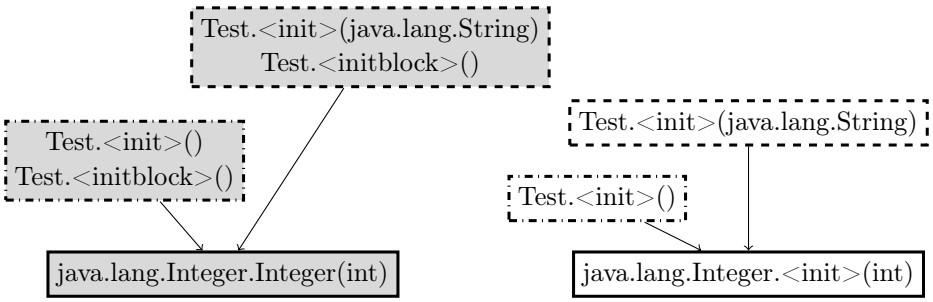



Figure 4: The basic name pairing in action: produced pairings

by each static analyzer tool. Every other cell in a row is a percentage that displays what percentage of the given tool's methods was found by the tool in the column.

Table 2: Results of the basic name pairing

| | Soot | OSA | SPOON | JCG | WALA | JDT |
|---|---|---|---|---|---|---|
| Soot | **4,022** | 51.24% | 52.76% | 57.46% | 57.36% | 50.97% |
| OSA | 24.4% | **8,446** | 100,00% | 96.39% | 80.88% | 91.13% |
| SPOON | 24.77% | 98.5% | **8,551** | 96.55% | 81.24% | 89.81% |
| JCG | 23.22% | 81.81% | 83.24% | **9,951** | 75.82% | 75.98% |
| WALA | 27.47% | 81.33% | 83.02% | 89.83% | **8,399** | 83.09% |
| JDT | 21.41% | 80.37% | 80.43% | 78.95% | 72.87% | **9,577** |

Looking at the table, it becomes apparent that the column of Soot contains quite low values. Soot found half of the methods compared to the other analyzers, therefore, its highest possible percentage is at about 50% - 60%. The reason for this discrepancy lies in the algorithmic differences between the tools, however, analyzing this is not the subject of the current paper.

## 4.3 Anonymous transformation

The basic name pairing cannot handle every Java language feature. One of them is the anonymous source code elements.

An anonymous class is an inner class without a name. It is useful when the programmer needs one instance of a class or interface with only certain overridden methods, so the actual subclass creation can be avoided. Lambda methods can be considered anonymous, however, most analyzers denote them with their interface name. Anonymous source code elements have a non-standardized, compiler generated name, meaning that static analyzers can name the same code element differently. Inner classes have a '$' sign in their name appended right after the name of the outer class. The '$' sign is followed by the name of the inner class.

In case of anonymous classes, a number is present after the '$' sign, however, the numbering is not consistent among the compilers and analyzer tools. Both global, project-wise numbering, and class level numbering is possible. The order of the numbering can also make a difference in the output of the tools. It is clear that our basic pairing approach that was introduced in the previous subsection is not sufficient for pairing anonymous code elements.

The transformation simply means that during the name unification process we replace the varying number after the '$' sign with a constant string. This means that multiple anonymous elements in a class will be aggregated into one, which is the explanation of smaller method numbers in the diagonal of Table 3. For example, if a class has multiple anonymous classes, all of them will be transformed for the unified anonymous class, causing a loss in the accuracy of the pairing. For projects that do not rely on anonymous classes very much - i.e. in a class there is at most one anonymous element - this heuristical approach is acceptable.

```
class AnonymousTest{
  public void print(){
    //...
  }
}

class Test{
  public static void main(String args[]){
    AnonymousTest t1 = new AnonymousTest(){
      @Override
      public void print(){
        //...
      }};

    AnonymousTest t2 = new AnonymousTest(){
      @Override
      public void print(){
        //...
      }};
  }
}
```

Figure 5: Sample code containing two anonymous classes

Figure 5 shows an example code containing two anonymous classes. Figure 6 portrays a call graph constructed from this code (left side) and the class-level aggregation of anonymous code elements after the anonymous transformation (right side). If this code snippet is part of a larger project, then the two anonymous classes may not have the same numbering in all of the produced call graphs. However, after the aggregation the unified anonymous nodes can be paired.

Table 3, that is constructed similarly to Table 2, shows the results of the method pairing improved with anonymous transformation. The green cells highlight those percentages that are higher compared to Table 2.

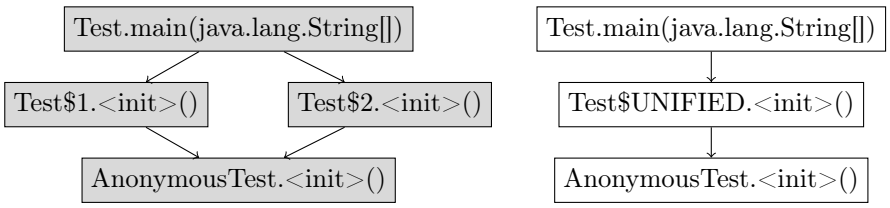



Figure 6: The process of anonymous transformation. The original graph is on the left, the transformed version is on the right side

Table 3: Results of the anonymous transformation

| | Soot | OSA | SPOON | JCG | WALA | JDT |
|---|---|---|---|---|---|---|
| Soot | **3897** | 51.78% | 53.09% | 56.58% | 56.38% | 54.68% |
| OSA | 24.23% | **8329** | 100,00% | 96.96% | 81.23% | 94.38% |
| SPOON | 24.54% | 98.81% | **8406** | 97.25% | 81.68% | 93.42% |
| JCG | 22.56% | 82.61% | 83.94% | **9776** | 75.39% | 78.71% |
| WALA | 26.75% | 82.39% | 83.97% | 89.75% | **8212** | 86.79% |
| JDT | 22.58% | 83.3% | 83.46% | 81.54% | 75.52% | **9437** |

## 4.4 Employing line information

We concluded that the previous heuristical solution should and could be improved, so that anonymous source code elements could be paired independently. It was a self-evident idea to include the line information in order to improve the accuracy of the method matching. However, we soon found out that the line information does not provide a perfect solution for the problems of method pairing because it is not as consistent among static analyzers as it was expected.

One obvious difficulty is that some of the tools process the source files themselves, while others work on the already compiled class files. Source code analyzers provide line information to the beginning and end of the method declaration. Byte code analyzers give the line information for the first statement of the method in question. In case of an empty method, the line information of the ending of the method declaration is present. This difference can be overcome by interval testing. Moreover, not every method has line information because they are compiler generated or they are part of the Java library. In other cases, only some of the tools can provide line information for a method. In addition to these difficulties, we realized that in a few cases tools provide the line information of the beginning of the class definition for some methods. These are inherited methods, whose return type was specialized by the child class (as it was described in the beginning of Section 4.2). As a consequence, some methods that certainly differ have the same line information.

Seeing these difficulties, it is clear that we cannot rely on line information

blindly, because it would misguide the pairing mechanism. Therefore, the usage of line information was restricted only for anonymous and generic source code elements, whilst, for traditional methods, the name-wise pairing was used. The challenge of anonymous elements has already been discussed. In their case, we used only the line information for matchmaking. Generic elements raised a new type of issue that is introduced in the next section.

Figure 7 depicts the pseudocode of the line information based anonymous pairing. The condition on line 23 is true if the package names of the two methods are equal, and the class and method names only differ after the $ sign (anonymous

```
1  /*
2  The method returns true if two nodes (methods) considered equal
       according to their line information
3  */
4  func checkLineInfo(m1,m2)
5    if m1.endLine NOT valid
6      m1.endLine=m1.startLine
7    end
8
9    if m2.endLine NOT valid
10     m2.endLine=m2.startLine
11   end
12
13   return (m1.startLine <= m2.startLine AND m1.endLine >= m2.endLine
         ) OR (m1.startLine >= m2.startLine AND m1.endLine <= m2.
         endLine)
14 end
15
16 /*
17 This method returns true if the given nodes (methods) are
       considered equal otherwise false
18 */
19 func anonymousPairing(m1, m2) //m1 and m2 are anonymous methods
20 begin
21   isEqual=false
22   if line-info available
23     if m1 differs m2 only in anonymous names
24       if checkLineInfo(m1, m2)
25         for i in m1.parameterCount
26           if m1.param[i] NOT equals m2.param[i]
27             return false
28           end
29         end
30         isEqual=true
31       end
32     end
33   end
34   return isEqual
35 end
```

Figure 7: The pseudocode of the line information based anonymous pairing

part). On line 24 we check if the two methods are the same according to the line information. Byte code analyzers provide the line information of the first statement of the examined method, while source code analyzers detect the method declaration. Moreover, some tools consider the comment in front of a method as part of its declaration, making the line-information of a method even more diverse. Method `checkLineInfo` depicts how the interval checking of the line information of two methods is done. If a tool does not provide end line number for the methods it will be initialized with the number of the start line. The equality of the parameter lists is examined on line 25-29, although, it is not necessary if we consider the provided line information valid.

Table 4 shows the improvement of results compared to the basic name-wise pairing that is summarized in Table 2. In case of Soot and WALA we can see a slight decrease in the number of methods. It is because these tools - erroneously - provided the same line information for some anonymous methods, therefore, they could not be handled separately. The approach best improved the pairing of the JDT as this is the most reliable tool for providing line information.

Table 4: Results of the transformation based on line information (anonymous)

| | Soot | OSA | SPOON | JCG | WALA | JDT |
|---|---|---|---|---|---|---|
| Soot | **3,976** | 51.84% | 53.37% | 56.97% | 56.87% | 54.83% |
| OSA | 24.4% | **8,446** | 100,00% | 96.39% | 80.88% | 93.27% |
| SPOON | 24.77% | 98.5% | **8,551** | 96.55% | 81.24% | 92.13% |
| JCG | 22.76% | 81.81% | 83.24% | **9,951** | 75.19% | 77.96% |
| WALA | 27.12% | 81.95% | 83.65% | 89.76% | **8,336** | 86.32% |
| JDT | 22.76% | 82.26% | 82.5% | 81.01% | 75.14% | **9,577** |

## 4.5 Strategy for handling generic elements

As the previous subsection indicated, generic source code elements need extra consideration during the pairing mechanism. Java generic classes and methods were introduced in JDK 5.0. They allow programmers to specify a set of methods and a set of types with only one method and class declaration, respectively. A single generic method can be called with arguments of various types. One important trait of generic classes is that they can be parameterized differently during instantiation. Generic type parameters can be bounded, which restricts the types that are allowed to be passed.

Static analyzers represent generic elements in the call graph in various ways. Table 5 shows the diversity of representations after the method name unification. It can be seen that the tools represent them with varying accuracy. Sometimes generic parameters are represented by the prototype that is present in the declaration, optionally involving the type restriction too (e.g., SOOT). In other cases, the type of the actual parameter is used, that is, the tool represents the same generic method

with multiple nodes but with differing generic parameters.

Table 5: Various representations of a generic method

| | |
|---|---|
| Declared method | `<T, K extends Child2> Generic2<Child2, Generic1<Child2> > methodGen(K c, Generic1<K> g, Class<?>...objects)` |
| Usage | `methodGen(new Child2(), new Generic1<Child2>(), Integer.class)` |
| Representations | |
| JCG | `methodGen(Child2,Generic1,java.lang.Class[])` |
| WALA | `Generic2 methodGen(Child2,Generic1,java.lang.Class)` |
| OSA | `Generic2 methodGen(Child2,Generic1,java.lang.Class)` |
| Soot | `Generic2 methodGen(Child2,Generic1,java.lang.Class[])` |
| SPOON | `methodGen(K extends Child2,Generic1,java.lang.Class[])` |
| JDT | `Generic2<Interface,Generic1> methodGen(K,Generic1<Interface>, java.lang.Class<?>[])` |

The ideal solution would be to pair the corresponding generic methods to each other, but because of the variety of the notations, matching them only through the basic pairing process caused inaccuracies. Although the package, class, and method names are the same, even the number of parameters are the same, the type of the parameters can differ. Unlike in the case of anonymous methods, it is not always possible to decide whether a generic method is generic or not, based on its name alone. Therefore, the line information is needed to decide if two methods with the same name and number of parameters correspond to the same generic method. If the line information is the same as well, then the two nodes apply to the same generic method. This heuristical assumption has a threat to validity if the tool provides false line information. What is more, the pairing is not possible if no line information is given. The pseudocode of the pairing algorithm for generic elements is shown in Figure 8. The `checkLineInfo` method is the same as in Figure 7. The heuristical method for matching the generic parameters is on line 11-15. As our pairing approach currently does not utilize the class hierarchy of the analyzed project, only a conservative matching is allowed with generic wildcards such as `K,T,E...` and with `java.lang.Object`, which is a base class for every other class. The reason for this conservative solution is that we want to avoid accidental matching of overridden methods. The manual validation proved this approach to be sufficient. Combining line information with generic elements caused another type of problem, which is summarized in Figure 9.

Figure 9 shows two static analyzers, Tool 1 and Tool 2 (denoted by grey ellipses) and methods they detected during analysis (denoted by white ellipses). The analyzed source code contains a generic method, `<T> void goo(T t)` and two normal methods, `void foo(int a, int b)` and `void foo(int a)`. Tool 1 represents `goo` in the call graph only with one node. As there is no restriction on the type, the tool denotes the parameter type as an `Object`. In contrast to this, Tool 2 associates three nodes to method `goo` based on the type of the actual parameters it was called with. All `goo` nodes have the same line information. The matching of the `foo`

```
1
2  /*
3  This method returns true if the given nodes (methods) are
       considered equal otherwise false
4  */
5  func anonymousPairing(m1, m2)
6  begin
7    isEqual=false
8    if line-info available
9       if checkLineInfo(m1, m2)
10        for i in m1.parameterCount
11          if (m1.param[i] equals m2.param[i] OR
12             m1.param[i] is java.lang.Object OR
13             m2.param[i] is java.lang.Object OR
14             m1.param[i] is GENERIC_WILDCARD OR
15             m2.param[i] is GENERIC_WILDCARD)
16            isEquals=true
17          else
18            isEquals=false
19            break
20          end
21        end
22      end
23   end
24   return isEqual
25 end
```

Figure 8: The pseudocode of the line information based generic pairing

methods is obvious, as both of them are represented with one node each. This is not the case with the pairing of method `goo`. The left side of the figure shows a possible matching of the nodes of Tool 1 to the nodes of Tool 2. The pairing of `goo` is denoted with a dashed line, as other matches would be possible if it was allowed to pair one method to multiple others. The right side of the figure shows the opposite direction: the matching of the nodes of Tool 2 to the nodes of Tool 1. It can be seen that all `goo` nodes will be paired to the same node in the graph of Tool 1, because there is no other option. As a consequence, there is asymmetry in the results depending on the direction from which we start pairing the nodes.

This described pairing anomaly can be resolved in multiple ways. One solution is to use the results as they are, without any further modifications. This approach emphasizes the differences between the tools' capabilities. Another option is to keep only those node-matchings that can be found from both directions. Finally, we can collect every possible pairing from both directions and put them into a union. The union pairing was the solution we decided to use. Table 6 summarizes the results of this approach. The structure of the table is similar as before, the green cells highlight the higher percentages compared to Table 4. There is a decrease in the number of methods because we counted the corresponding generic methods as one.

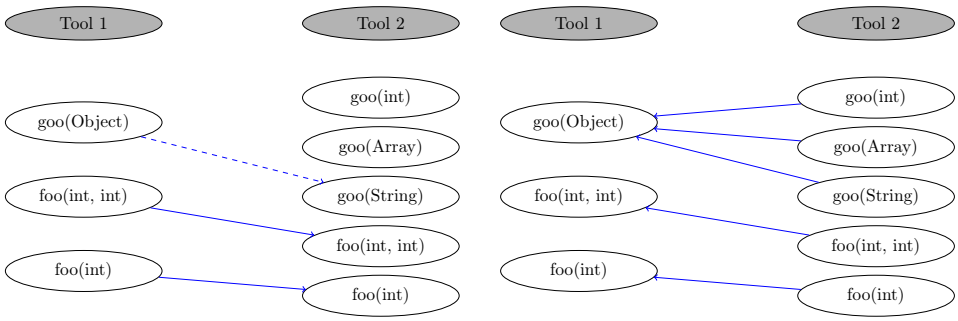



Figure 9: Pairing anomaly

Table 6: Results of the transformation based on line information (anonymous and generic elements)

| | Soot | OSA | SPOON | JCG | WALA | JDT |
|---|---|---|---|---|---|---|
| Soot | **3,969** | 51.85% | 53.39% | 56.97% | 56.97% | 55.1% |
| OSA | 24.38% | **8,441** | 100,00% | 96.39% | 80.9% | 95.53% |
| SPOON | 24.75% | 98.5% | **8,545** | 96.55% | 81.26% | 94.36% |
| JCG | 22.76% | 81.93% | 83.36% | **9,928** | 75.18% | 79.97% |
| WALA | 27.16% | 81.99% | 83.69% | 89.74% | **8,332** | 88.02% |
| JDT | 23.1% | 85.83% | 86.07% | 84.52% | 77.05% | **9,547** |

## 4.6 Remaining differences

Table 6 shows that we could not achieve 100% pairing for the tools, a significant number of nodes remained unmatched. We manually investigated the root causes for this, in order to find possible ways to improve our pairing mechanism. However, our in-depth examination revealed that most of the unmatchings cannot be resolved. The reasons for the differences can be categorized as follows:

- A tool detects a method type that other tools do not represent, therefore some nodes will not have images in the other tools' graph.
  - Soot represents much more static initializer nodes then other tools.
  - WALA places more Java library nodes and calls into the generated call graphs.
  - SPOON represents Java static field initialization with a unique node.
- The methods that can be found in the bytecode slightly differ from the methods of the source code.

  - Bytecode analyzers (JCG, WALA, Soot) find compiler generated `access$XXX` methods, which cannot be paired with improper line information.
  - Source code analyzers detect only default constructors for `Enum` classes. Byte code analyzer tools represent the valid constructors with an `Integer` and a `String` parameter.
  - In the compiled sources, the methods of inner classes have an extra parameter, a reference to the outer class. This parameter is missing from the findings of the source code analyers.

- There are algorithmic differences in the handling polymorphic calls.
  - Tools that employ less accurate analysis techniques represent more interface and base class methods instead of the methods of the subclasses.
  - JCG represents inherited methods as the method of the child class, while other tools represent them as part of the base class.
- Methods that do not have at least one method call are excluded. OSA and SPOON have this feature.
- Line information for anonymous and generic methods is missing.

We concluded from our findings that our pairing mechanism could only be improved with more reliable line information.

## 4.7 Edge similarity

Based on the implemented node pairing mechanisms, the pairing of the edges was also performed. Two edges are considered to be a pair if their endpoints are matched with each other. If one or both nodes of an edge are unmatched, then the edge itself is considered to be pairless too. This subsection discusses how the improvement of the node pairing affects the number of edges that can be paired with each other.

Table 7 and Table 8 present the call edge comparison results of the Commons Math project. Their structure is similar to the previous tables': the diagonal elements contain the number of calls detected by each tool, while every other cell in a row shows how many percent of the tool's calls were found by the tools in the columns. Table 7 corresponds to the basic name pairing mechanism and Table 8 shows the results achieved by using our final approach. Higher percentages are highlighted with green. A slight decrease can be observed in the number of call edges. As Table 2 and Table 6 show, some of the nodes were aggregated, and, because of this, a few duplicated edges were eliminated.

As expected, there are improvements in the number of successfully paired call edges, although the change is not really significant. Even if we take into account that there are possibly pairable methods, there are considerably low pairing ratios. This suggests that there are vital differences in the topology of the call graphs as well. The sampling of the unmatched call edges supports this assumption, however, a more in depth examination is needed to make further conclusions.

Table 7: Edge similarity using the basic name pairing method

| | Soot | OSA | SPOON | JCG | WALA | JDT |
|---|---|---|---|---|---|---|
| Soot | **28,542** | 12.46% | 12.72% | 14.01% | 13.39% | 11.57% |
| OSA | 17.73% | **20,059** | 99.83% | 88.92% | 58.03% | 83.25% |
| SPOON | 17.71% | 97.67% | **20,501** | 87.70% | 57.98% | 82.28% |
| JCG | 17.52% | 78.14% | 78.77% | **22,826** | 53.18% | 66.94% |
| WALA | 23.36% | 71.14% | 72.65% | 74.19% | **16,363** | 62.08% |
| JDT | 17.1% | 86.52% | 87.4% | 79.16% | 52.63% | **19,302** |

Table 8: Edge similarity using the final approach

| | Soot0 | OSA0 | SPOON0 | JCG0 | WALA0 | JDT0 |
|---|---|---|---|---|---|---|
| Soot0 | **28,485** | 12.48% | 12.74% | 13.72% | 13.28% | 12.14% |
| OSA0 | 17.72% | **20,057** | 99.83% | 88.9% | 58.03% | 90.78% |
| SPOON0 | 17.7% | 97.67% | **20,499** | 87.68% | 57.98% | 89.82% |
| JCG0 | 17.13% | 78.14% | 78.77% | **22,817** | 52.83% | 72.29% |
| WALA0 | 23.18% | 71.33% | 72.84% | 73.86% | **16,319** | 65.23% |
| JDT0 | 17.92% | 94.4% | 95.46% | 85.51% | 55.19% | **19,288** |

# 5 Comparison with a topology-based algorithm

In this Section we describe a comparison with the neighbor matching algorithm introduced in Section 2. Our goal was to study how a neighborhood-based algorithm performs in terms of accuracy and computational time compared to our approach.

## 5.1 Utilizing the topology-based method

We downloaded the C++ implementation [7] of Nikolič's work [25]. Only output formatting modifications were applied. We transformed the call graphs that were built by the six call graph creator tools so the iterative tool could take them as an input. The iterative tool requires only the call edge information, no node labeling is needed.

Like other iterative graph similarity algorithms, this one also produces a similarity matrix over the nodes of the compared graphs. Nikolič's innovation was the normalization of the similarity values between 0-1, so that the higher values indicate greater similarity. We computed and processed the similarity matrices of the examined projects for each call graph creator tool pair.

[7] `http://www.matf.bg.ac.rs/~nikolic/software.html`

## 5.2 Evaluation of results

To interpret the similarity values as pairings of nodes we searched for the maximum values in each row and column. Although this seems like a straightforward solution the similarity values were rather noisy, meaning that in many cases there were multiple similarity values around the maximum of a row or a column. Let us consider the similarity matrix of Soot and SPOON produced from their call graphs for the Commons Math project. If we pick a method by random there is a high chance that its pair defined by the maximum will be a noisy result. For example, in case of the `org.apache.commons.math3.util.FastMathLiteralArrays.loadExpFracB()` method which was detected by SPOON and has valid line information the highest similarity value is around 0.6. This corresponds to the following method: `org.apache.commons.math3.transform.FastFourierTransformer$MultiDimensionalComplexMatrix.<init>(java.lang.Object)`. It is clear that this pairing is invalid. To reduce the tremendous noise, we decided to take a pairing into consideration only when it is supported by both the column and row point of view, meaning that the value is both a column and a row maximum (ceratin matchings). If we examine the previous Soot-SPOON comparison this way we reduced the number of pairings reported only by the iterative approach from 12255 to 252. As it can be calculated from Table 6, our attempt detects 2120 pairings in the SPOON - Soot call graph comparison. There are 77 matches of the iterative tool for the SPOON - Soot comparison, which were also detected by our algorithm. If we consider the uncertain maximums as well, this number is 239. These matches were found valid, meaning that out of the 252 certain pairs of the Soot - SPOON comparison about 30% is valid.

Our main interest was to analyze the validity of pairings detected only by the iterative method. There were 2100 unique pairings out of the 15 pairwise comparisons of the 6 call graphs created for the Commons Math project. The manual investigation showed that 2036 of them were invalid, whilst 64 were valid, which is about 3%. The valid matchings had a specific characteristic. All of them were generated constructors of anonymous classes. Byte code analyzer tools represent these constructors with precise parameter lists containing references to the outer class and to the local fields used in the body of the anonymous class as well, while source code analyzers detect only the parameters that can be found in the sources. Naturally, without proper line information and with differing parameter lists, our node pairing mechanism is doomed to fail on these type of methods. The error could be resolved if the source code position of these constructor methods would be associated at least with the declarataion of the anonymous class, and by loosening our requirement for entirely equal parameter lists. It has to be noted that even the manual validation failed in a very few cases, when there was no line information for one member of the pair and the outer class contained multiple anonymous classes.

### 5.3 Summarization

The manual investigation showed that the results are similar for the other projects as well. In case of the Joda-Time project[8], only 3 pairings were valid out of 754. We concluded from our findings that on average the validity of the pairings that are found only by the iterative method less than 10%. Section 4.7 indicates that there are significant differences in the number and type of detected call edges, which can be a reason for the noisiness of this topology-based method.

The comparison with a topology-based method revealed a very specific weak point of our pairing mechanism. Although the problem is limited to a little subset of the nodes, it still has to be addressed in the future. If the static analyzer tools would provide line information for these anonymous initializers, for instance by associating them with the position of the declaration of the anonymous class, our approach could pair them. It would be a straightforward idea to combine our approach with this iterative method to resolve the described problem. However, the expansion would not be trivial, especially if we consider that our pairing mechanism took 12 minutes, while the iterative algorithm finished the Commons Math project over 17 hours.

Despite the problems of anonymous constructors, this comparison assured us that we find no matches that a neighbor-based algorithm would not and we miss a high percentage of noisy results that the iterative method reports. Moreover, the computation time of our pairing mechanism does not scale with the size of the input graphs as badly as that of the iterative method. On small sample graphs both implementations finish within seconds, however in case of projects with a few thousands lines of code the iterative method needs hours compared to the couple of minutes that our approach requires.

## 6 Conclusions

In the future, we plan to compare the capabilities of static call graph creator tools. This could be done by comparing what methods and calls are present in the generated call graphs. If the nodes of the call graphs are matched, then comparing the calls is a straightforward task. That is the reason why we paid so much attention to the unifying process of the methods. This paper was a necessary preliminary work for the upcoming quality comparison of the tools.

We collected and, where necessary, modified six Java static analyzer tools to generate call graphs for multiple large projects. By investigating the resulting graphs, we realized that the unification of method names is needed, in order to be able to match the corresponding nodes to each other. The unification process - and hence the pairing mechanism - has been refined in several steps. We highlighted two common language elements, the anonymous and generic methods that needed careful consideration and made the improvement of the process necessary. Multiple solutions were proposed. One heuristical - but less accurate - approach for

[8] `https://github.com/JodaOrg/joda-time`

anonymous elements is the anonymous transformation. However, with line information they could be handled better, along with the generic code elements. We performed a manual validation of the different pairing strategies on a sample code, containing all features of Java 8. The source and the results are available in the online appendix. The results of the large projects were also manually investigated. Our solution was compared to a topology based node pairing algorithm as well.

In our final solution, we used the basic name-wise pairing for normal methods, line information-based pairing for anonymous methods and a combined solution for generic methods. In this combined solution, if two methods have the same package, class and method name, have the same number of parameters and have the same line information, it is assumed that they correspond to the same generic method declaration. The analyzers may represent the same generic method with different number of nodes in their call graphs. This asymmetry was solved by collecting every possible pairing between these nodes.

The manual validation proved that better pairing could be achieved if we could acquire more accurate line information of the methods. However, the reason for matchless nodes lies in the differences of the static call graph creators themselves, therefore, the matching of some nodes is impossible.

# References

[1] Ahmad Bhat, Sajad. A practical and comparative study of call graph construction algorithms. *IOSR Journal of Computer Engineering*, 1:14–26, 01 2012. DOI: 10.9790/0661-0141426.

[2] Andersen, Lars Ole. Program analysis and specialization for the C programming language. Technical Report May, 1994. 10.1.1.109.6502.

[3] Apache BCEL Home Page. https://commons.apache.org/proper/commons-bcel.

[4] Bacon, David F. and Sweeney, Peter F. Fast Static Analysis of C++ Virtual Function Calls. *SIGPLAN Not.*, 31(10):324–341, October 1996. DOI: 10.1145/236338.236371.

[5] Blondel, Vincent, Gajardo, Anahi, Heymans, Maureen, Senellart, Pierre, and Van Dooren, Paul. A measure of similarity between graph vertices: Applications to synonym extraction and web searching. *SIAM Review*, 46:647–666, 12 2004. DOI: 10.2307/20453570.

[6] Call Hierarchy Printer GitHub Page. https://github.com/pbadenski/call-hierarchy-printer.

[7] Champin, Pierre-Antoine and Solnon, Christine. Measuring the similarity of labeled graphs. In Ashley, Kevin D. and Bridge, Derek G., editors, *Case-Based Reasoning Research and Development*, pages 80–95, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[8] Christodorescu, Mihai and Jha, Somesh. Static analysis of executables to detect malicious patterns. In *Proceedings of the 12th Conference on USENIX Security Symposium - Volume 12*, SSYM'03, pages 12–12, Berkeley, CA, USA, 2003. USENIX Association.

[9] Dean, Jeffrey, Grove, David, and Chambers, Craig. Optimization of Object-Oriented Programs Using Static Class Hierarchy Analysis. In Tokoro, Mario and Pareschi, Remo, editors, *ECOOP'95 — Object-Oriented Programming, 9th European Conference, Åarhus, Denmark, August 7–11, 1995*, pages 77–101, Berlin, Heidelberg, 1995. Springer Berlin Heidelberg.

[10] Eclipse JDT Home Page. `http://www.eclipse.org/jdt/`.

[11] Eclipse JDT Home Page. `www.eclipse.org/eclipse/`.

[12] Eshera, M. A. and Fu, K. A graph distance measure for image analysis. *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-14(3):398–408, May 1984. DOI: `10.1109/TSMC.1984.6313232`.

[13] Feng, Yu, Anand, Saswat, Dillig, Isil, and Aiken, Alex. Apposcopy: Semantics-based detection of android malware through static analysis. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 576–587, New York, NY, USA, 2014. ACM. DOI: `10.1145/2635868.2635869`.

[14] Grove, David and Chambers, Craig. A framework for call graph construction algorithms. *ACM Trans. Program. Lang. Syst.*, 23(6):685–746, November 2001. DOI: `10.1145/506315.506316`.

[15] Grove, David, DeFouw, Greg, Dean, Jeffrey, and Chambers, Craig. Call Graph Construction in Object-oriented Languages. In *Proceedings of the 12th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '97, pages 108–124, New York, NY, USA, 1997. ACM. DOI: `10.1145/263698.264352`.

[16] Heymans, Maureen and Singh, Ambuj K. Deriving phylogenetic trees from the similarity analysis of metabolic pathways. *Bioinformatics*, 19 Suppl 1:i138–46, 2003.

[17] Java Call Graph GitHub Page. `https://github.com/gousiosg/java-callgraph`.

[18] JavaParser - for processing Java code Homepage. `https://javaparser.org/`.

[19] Koutra, Danai, Parikh, Ankur, Ramdas, Aaditya, and Xiang, Jing. Algorithms for graph similarity and subgraph matching. 02 2019.

[20] Lhoták, Ondrej. Comparing call graphs. In *ACM SIGPLAN/SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, pages 37–42, 2007.

[21] Lhoták, Ondřej and Hendren, Laurie. Context-Sensitive Points-to Analysis: Is It Worth It? In Mycroft, Alan and Zeller, Andreas, editors, *Compiler Construction*, pages 47–64, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[22] Macindoe, O. and Richards, W. Graph comparison using fine structure analysis. In *2010 IEEE Second International Conference on Social Computing*, pages 193–200, Aug 2010. DOI: 10.1109/SocialCom.2010.35.

[23] Melnik, Sergey, Garcia-Molina, Hector, and Rahm, Erhard. Similarity flooding: A versatile graph matching algorithm and its application to schema matching. pages 117 – 128, 02 2002. DOI: 10.1109/ICDE.2002.994702.

[24] Murphy, Gail C., Notkin, David, Griswold, William G., and Lan, Erica S. An Empirical Study of Static Call Graph Extractors. *ACM Trans. Softw. Eng. Methodol.*, 7(2):158–191, April 1998. DOI: 10.1145/279310.279314.

[25] Nikolić, Mladen. Measuring similarity of graph nodes by neighbor matching. *Intell. Data Anal.*, 16(6):865–878, November 2012. DOI: 10.3233/IDA-2012-00556.

[26] OpenStaticAnalyzer GitHub Page. https://github.com/sed-inf-u-szeged/OpenStaticAnalyzer.

[27] Pawlak, Renaud, Monperrus, Martin, Petitprez, Nicolas, Noguera, Carlos, and Seinturier, Lionel. Spoon: A Library for Implementing Analyses and Transformations of Java Source Code. *Software: Practice and Experience*, 46:1155–1179, 2015. DOI: 10.1002/spe.2346.

[28] Sable *J Home Page. http://www.sable.mcgill.ca/starj/.

[29] Sable/Soot GitHub Page. https://github.com/Sable/soot.

[30] Sanfeliu, A. and Fu, K. A distance measure between attributed relational graphs for pattern recognition. *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-13(3):353–362, May 1983. DOI: 10.1109/TSMC.1983.6313167.

[31] Sundaresan, Vijay, Hendren, Laurie, Razafimahefa, Chrislain, Vallée-Rai, Raja, Lam, Patrick, Gagnon, Etienne, and Godin, Charles. Practical virtual method call resolution for java. *SIGPLAN Not.*, 35(10):264–280, October 2000. DOI: 10.1145/354222.353189.

[32] Tip, Frank and Palsberg, Jens. Scalable propagation-based call graph construction algorithms. In *Proceedings of the 15th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '00, pages 281–293, New York, NY, USA, 2000. ACM. DOI: 10.1145/353171.353190.

[33] Tversky, Amos. Features of similarity. *Psychological Review*, 84(4):327–352, 1977. DOI: 10.1037/0033-295X.84.4.327.

[34] Ullmann, J. R. An algorithm for subgraph isomorphism. *J. ACM*, 23(1):31–42, January 1976. DOI: 10.1145/321921.321925.

[35] Wagner, Tim A., Maverick, Vance, Graham, Susan L., and Harrison, Michael A. Accurate static estimators for program optimization. *SIGPLAN Not.*, 29(6):85–96, June 1994. DOI: 10.1145/773473.178251.

[36] WALA Home Page. http://wala.sourceforge.net/wiki/index.php/Main_Page.

[37] Weiser, Mark. Program slicing. In *Proceedings of the 5th International Conference on Software Engineering*, ICSE '81, pages 439–449, Piscataway, NJ, USA, 1981. IEEE Press.

[38] Wicker, Nicolas, Nguyen, Canh Hao, and Mamitsuka, Hiroshi. A new dissimilarity measure for comparing labeled graphs. *Linear Algebra and its Applications*, 438(5):2331 – 2338, 2013. DOI: 10.1016/j.laa.2012.10.021.

[39] Zager, Laura A. and Verghese, George C. Graph similarity scoring and matching. *Applied Mathematics Letters*, 21(1):86–94, jan 2008. DOI: 10.1016/j.aml.2007.01.006.

Acta Cybernetica **24** (2019) 157–172.

# Combining Common Sense Rules and Machine Learning to Understand Object Manipulation*

András Sárkány[ab], Máté Csákvári[ac], and Mike Olasz[c]

**Abstract**

Automatic situation understanding in videos has improved remarkably in recent years. However, state-of-the-art image processing methods still have considerable shortcomings: they usually require training data for each object class present and may have high false positive or false negative rates, making them impractical for general applications. We study a case that has a limited goal in a narrow context and argue about the complexity of the general problem. We suggest to solve this problem by including *common sense rules* and by exploiting various state-of-the art *deep neural networks* (DNNs) as the detectors of the conditions of those rules.

We want to deal with the manipulation of unknown objects at a remote table. We have two action types to be detected: 'picking up an object from the table' and 'putting an object onto the table' and due to remote monitoring, we consider monocular observation. We quantitatively evaluate the performance of the system on manually annotated video segments, present precision and recall scores. We also discuss issues on machine reasoning. We conclude that the proposed neural-symbolic approach a) diminishes the required size of training data and b) enables new applications where labeled data are difficult or expensive to get.

**Keywords:** situation understanding, event recognition, computer vision

## 1 Introduction

When we talk about situation understanding in AI, we usually imagine a scenario with people acting in front of a camera and the task of the computer system is to assign categories to the ongoing events. Our work presented in this paper is about detecting one particular example of such events: an object being lifted up from or put down on a table. In most cases the need to recognize such a scenario does not

*This research has been supported by the European Union, co-financed by the European Social Fund (EFOP-3.6.3-VEKOP-16-2017-00002).

[a]Faculty of Informatics, Eötvös Loránd University, Budapest, Hungary.

[b]Present address: Argus Cognitive Inc., USA

[c]Argus Cognitive Inc., USA

DOI: 10.14232/actacyb.24.1.2019.11

stand alone, rather it comes from a higher level goal or some kind of application logic. Looking at this problem from a top-down perspective, it is generally true that fulfilling one such higher level goal requires multiple lower level recognition tasks to be solved and this hierarchy can go multiple levels.

We start from the concept of spatio-temporal pattern recognition that we will use to conceptualize the complexity of such tasks. Pattern recognition in general, is the process of analyzing data and making a decision, such as classifying a sample to different categories, with the help of known regularities in the data [5]. In our case, pattern recognition concerns spatio-temporal ones that we shall call event recognition in the following.

In the case of classification we expect the data space to have 2 properties: 1) the data points corresponding to a category are confined to a region of the space possibly having lower effective dimensionality 2) the data space is locally smooth around these confined regions, meaning that data points close to each other usually belong to the same category. These concepts are analogous to the generalization property. However the data space that we work in not necessarily holds these properties but with the help of highly nonlinear transformations such a space can be constructed. This is especially true in real-world applications where data comes from sensors and it is represented in high dimensional space, like images.

Known regularities encoded in the machine can have two sources: a) human knowledge about the data and b) a model trained by data with an algorithm. Humans can comprehend and give rules for categories for a data space of maximum 2-3 effective dimensions if it has the confinement and smoothness properties. Unsupervised machine learning algorithms can perform nonlinear dimension reduction, but for most real world problems supervised methods are needed, which require training labels. (Also high dimensional data requires more advanced algorithms.)

Annotated training data is the most expensive component of the pattern recognition process and its size rests on two main factors. First, high dimensionality of the input exponentially increases the required training data, also known as the curse of dimensionality [4].

Secondly, special target categories are less frequent in real life therefore collecting the necessary amount of data requires more resources (time, money). Besides, finding the regularities that characterize a special category in comparison to a more general one from the same input requires more training data.

Consequently end-to-end machine learning from high dimensional data to very special categories requires the most expensive kind of data, and the most advanced algorithms.

Our scenario, and other examples of situation understanding, fit the description above: the video recordings we work with, consisting of $\geq$1 million 3-channel pixels for 25 fps, make a very high dimensional input. Collecting appropriate training data that contains the variance of the possible samples requires many subjects in many different scenarios from different viewpoints.

In our scenario the scene is recorded by a monocular RGB camera. Algorithm development for such devices is motivated by the inexpensiveness and commonness of them and it opens a path for a wide range of applications.

Lifting up or putting down an object viewed from an arbitrary (but not maliciously) placed camera is exposed to ambiguity even for human viewers of the scene because of a) occlusion of body parts and objects (to add to the confusion occlusion is caused by other body parts and objects) b) other type of hand movements similarly executed c) illusory perceptions caused by the information loss of the camera projection. So as data is the biggest obstacle how can we save on the costs of data acquisition?

We propose a process of breaking down a difficult event recognition task, which is only solvable with very expensive data, into smaller problems that we can solve with less resource such as freely available data and human knowledge. To the best of our knowledge, this work is the first one that offers a non-obtrusive automated solution to the quantification of picking up and putting down objects.

We describe our approach and the actual recognition pipeline implemented in this paper in Section 2. We compared our method to ground truth annotations on videos provided by Rush University Medical Center showing Autism Diagnostic Observation Schedule, Second Edition (ADOS-2) sessions with different patients. We report the results of our evaluation in Section 3. We discuss the results and provide possible improvements for future work in Section 4.

# 2 Method

In this section we describe our methodology for designing our pattern recognition pipeline. We refer to the related work Section 2.1. We outline the theoretical approach in Section 2.2. We used multiple machine learning models, algorithms and data trained or developed by other researchers, we describe these in Section 2.3. Our event recognition pipeline is depicted in Section 2.5.

## 2.1 Related work

There have been many advances in situation understanding and activity recognition in recent years and many datasets have been created for sets of action categories that can be used as benchmarks. In the early days the task was set as a classification task [9] but it moved to a detection task as methods evolved [18]. Recently most methods first extract sparse or dense timestamps and represent events by various descriptors, then use these to recognize similar space-time intervals. Some use strong video features such as histogram of gradients (HOG), motion history images (MHI) [2], others utilize pose estimation [18] or object detection [21]. There are supervised [8] and self-organizing solutions [3] as well, also wide usage of deep learning methods [19]. It is common in them that they use training data and they learn from samples of the event classes they aim to detect.

In contrast, our approach combines several detectors, put together by exploiting human knowledge and it doesn't require samples of the target event class, making it a viable alternative.

## 2.2 Approach

Our key idea is to transform our pattern recognition task to the composition of multiple "smaller" tasks. With the help of our human intuition we include a more general concept between the input data and our special category. The training data for this general concept is cheaper to manufacture than what we would have needed for our original task. The representation of this general concept has significantly less dimensions than the dimensionality of the original input data, lowering the required training data size for the second recognition task, which is identifying the special concept in the general one.

For example our recognizable action, lifting up an object, is a special case of a pose time series data of a person, so by utilizing a pose estimation algorithm like the convolutional PoseMachine [23], we include a general concept, Pose series, and we can define a new pattern recognition task where the input is the representation for this new concept and our target category is our original target category. Pose series will be represented by 17 joints, each with 2 dimensions multiplied by the length of the series in image frames which is an enormous decrease compared to the dimensionality of the original video data space.

This decomposition can be done arbitrary number of times introducing multiple intermediate concepts. Each decomposition replaces a pattern recognition task with two easier one in terms of data acquisition. Since information is lost when adding an intermediate concept, "sibling" concepts that are between the same input- and target data representations are necessary components. Eventually intermediate concepts will comply to the important factors for the original recognition task.

Each recognition task can be solved in any possible way outlined previously: a) supervised machine learning b) unsupervised dimension reduction c) human rules to assign set of data to a target category. Only a) requires training data where a very beneficial possibility is to use off-the-shelf data (or even trained models) created by others for a general concept that can be used for one's target category as well. From the perspective of expensive data acquisition the coupling of b) and c) can turn the tide from impossible to possible.

We stated previously that humans can formulate rules for comprehensible 2-3d spaces and this knowledge acquisition can save the collection of training data. Dimension reduction on its own can't really solve pattern recognition but can produce the representation that is usable for human rule creation. If we can decompose the original task in a way that such intermediate recognition task that is solvable with dimension reduction and human rules emerges, we can omit a data acquisition step.

The development of such a pipeline is an evolutionary process: we investigate the effect of including and excluding components through quantified metrics and qualitative analysis of samples of false positive and false negative recognitions. Consequently our pipeline have many other variations with other intuitive ideas.

## 2.3 External components

We briefly list the algorithmic components used in our pipeline. For further details please refer to the literature:

- **Convolutional Pose Machine**[23]: A deep learning method for human pose estimation. It provides estimation for key anatomical points of the human body. We use this method mainly to extract elbow and wrist coordinates so that we can further analyze the lower arm movement of the subject.
- **Mask-RCNN**[11]: A variant of the popular Faster-RCNN[17] object detection method. Other than providing bounding boxes for many object categories it also provides fine-grained instance segmentation of the underlying object.
- **Hand detector**: We used an R-FCN[6] based hand detector trained on a hand dataset [16].
- **Optical flow**: We used FlowNet2.0[12] as optical flow algorithm.

## 2.4 Target event description

We consider a scenario where a person interacts with objects on a flat surface, most likely a table viewed from a stationary camera. We decompose the event into three required subevents.

Putting down an object:

1. the person holds an object in her hand away from the future release location
2. the person puts it down on the surface
3. the person releases the object and moves her hand away from the release location

Picking up an object:

1. the object is on the table, and the person's hand is away from the object location
2. the person moves her hand towards the object and grabs it
3. the person takes the object away from its original position on the table

We introduced some limitations with these event definitions. We add three more to the list:

1. only events shorter than 10 seconds are taken into consideration
2. the object needs to be visible in the video for at least a few frames at the appropriate step (Put down 3., Pick up 1.)

3. the location of the object (without the object being there) needs to be visible for at least for a few frames at the appropriate step (Put down 1., Pick up 3.)

4. the required distance of the hand from the object location (pixel-wise on the video) is approximately the same size as a hand.

These limitations makes our task more special which means that there will be events that would fall into the category of picking up an object to a human observer but we don't consider the recognition of those.

## 2.5 Pipeline

As we described in Section 2.2 our goal is to build a pattern recognition pipeline that minimizes data acquisition which is extremely expensive in the case of a very special target category. We relied on our human intuition to decompose the original task and developed it in an evolutionary fashion.

Our main idea is twofold: when an object is moved there are two frames from the video of the stationary camera where the only difference of these images relevant parts determine the object's pixel representation; it "appears" or "disappears". We also note that grabbing or releasing an object involves a temporary stop of the hand's movement. We combine these two ideas to an algorithm where we first search for points in time when the hand is stopped, select these as candidates, then search one frame preceding and one following the candidate point, where the difference of the two frames shows the (dis)appearing object.

If a candidate is false, for example the person just rested her hand on the table for a second without moving any object, then the difference of the retrieved images will be zero, as both images show the part of the table unchanged.

We specify common sense assumptions and derive algorithmic components to:

1. select candidate positions

2. refine candidate positions by hand detection

3. select the two relevant frames

4. process the images to neglect irrelevant differences caused by interfering actions in the scene

5. take the image difference

In the following sections we describe each step in detail.

### 2.5.1 Candidate selection

We find timestamps of object grabbing by assuming that this action requires that the hand stops for at least an instant. Thus we look for such changes in the speed of the forearm which we measure by optical flow at regions estimated from elbow and wrist joint coordinates. If the magnitude of average velocity in that region

is at a local minima, then the timestamp is selected as a candidate for change detection. The local minimas are found by using gaussian filter on the velocity magnitude signal and then using a peak finding algorithm[7]. For each of the candidate position we assume that the appearing/disappearing object is occluded by the hand in the instant of releasing/grabbing.

#### 2.5.2 Candidate refinement

We refine the candidates obtained in the previous step by using an R-FCN based hand detection algorithm[6]. If the hand detection fails, meaning the hand cannot be find where the movement stopped then we discard this candidate position. A possible situation where this can happen is that the hand is under the table. We are using the bounding boxes obtained here as a RoI (region of interest) in the following steps.

#### 2.5.3 Image selection

In this step we select two frames, $I_{t_1}$ and $I_{t_2}$, that will be used at the image differencing step. We search backwards and forward in time to find frames where the object is not occluded by the hand, to find $I_{t_1}$ and $I_{t_2}$. This is done by simply checking if the bounding boxes of the hand moved significantly since the frame in which the object was grabbed. The search has a max duration parameter in both directions, if the hand doesn't leave the RoI in that time span, the candidate position is discarded. This parameter is set to 120 frames (4.8 seconds for a 25 fps video).

#### 2.5.4 Interference removal

We found that the selected frames and RoI given by previous steps contains differences other than the object we were looking for. These differences come from different sources: effect of actors interfering in the RoI (e.g. body parts, shadow, other manipulated object). To neutralize these effects we create a binary mask on the RoI that neglects pixels that belongs to these phenomena. In fact we estimate multiple binary masks with different strategies and take the union of the relevant pixels found by each method. These are the following:

#### 2.5.5 Interference removal - Body occlusion

We found that in many cases there were body parts inside the RoI which accounted for many of the changes that could be found between the two frames. We used Mask-RCNN[1][11] for filtering out pixels corresponding these parts.

#### 2.5.6 Interference removal - Long-term optical flow

There are changes between $I_{t_1}$ and $I_{t_2}$ that correspond to small movements over a longer period of time. A possible example is when the edge of the paper moved on

which the object was placed. To account for these differences we use optical flow between $I_{t_1}$ and $I_{t_2}$.

#### 2.5.7 Interference removal - Short-term optical flow

There can be ongoing actions inside RoI in $I_{t_1}$ and $I_{t_2}$ which we would also like to filter out. These events are typically hand movements or some other object movement. The optical flow between a frame at time instant $t$ and $t-1$ helps in removing any these effects inside the RoI (e.g. hand is still there but moving). This could be done for any $t$ and $t-k$ time instants, however we found $k=1$ to be sufficient.

#### 2.5.8 Image difference

We combine the binary masks obtained from previous steps to $I_{t_1}$ and $I_{t_2}$ then taking their difference as follows [13]:

$$I_{final}(x,y) = \| I_{t_1}(x,y) - \left( \frac{\sigma_1}{\sigma_2}(I_{t_2}(x,y) - \mu_2) + \mu_1 \right) \|_2$$

where $\mu_1, \sigma_1$ and $\mu_2, \sigma_2$ are the mean and standard deviation of $I_{t_1}$ and $I_{t_2}$ respectively.

After calculating the difference image, we perform the following steps to produce the final mask:

1. threshold $I_{final}$ to keep changed part of the images
2. drop small continuous blobs of the difference image for noise reduction

If the remaining covered area of the final mask is larger than 1% of the image area then we keep the difference as the object threshold, otherwise we discard the candidate position.

# 3 Evaluation

## 3.1 Data

We evaluated our pattern recognition pipeline on video segments that show a child taking part in an ADOS-2 diagnostic interview. This test includes different types of playful activities conducted by a clinician who is also present in the same room. The clinician interacts with the child during these games. Some of these tests contain objects that are moved from one place to another. We selected two activities "Puzzle game" and "Storytelling with toys" from three subjects. Segments for these games lasts 2-3 minutes and 10 minutes respectively. The dimensions of the puzzle pieces are around 5 cm x 3cm x 0.5cm colored blue and purple. The toys used for storytelling are dolls, plastic animals and tools of various sizes, their largest dimension ranges from 5 cm to 25 cm and colored diversely (examples can

(a) Selected image $I_{t_1}$ (b) Selected image $I_{t_2}$ (c) Final mask applied to $I_{t_1}$

Figure 1: A few examples from our evaluations. The algorithm starts by finding instants when a hand stops , see, column (b). Frames selected before and after the time instants of the hand stops are shown in columns (a) and (c), respectively. Column (d): masked images after applying the derived filters and taking the image differences. (See text for more information).

be seen on Figure 1). The room (size: 2.5 m x 2.5 m) has the same layout in all recordings with a single table, two chairs and optionally some cabinets holding the different tools for the different tasks. A single light source illuminates the room to standard indoor lighting.

Videos were recorded with a resolution of 1920x1080 pixels. Ground truth annotations of the precise temporal extent of the events were created with our own video annotator tool. Annotations were created and verified by our colleagues using a detailed description (Section 2.4), basically a set of rules about the relevant events, i.e. "picking objects up" and "putting objects down". We would like to emphasize that the ADOS-2 interviews were designed, conducted and recorded independently from our work, meaning that the properties of our proposed system had no effect on how the data was collected.

Statistics of each video segment's length and ground truth coverage is on Table 1.

Table 1: Length of input videos and coverage of ground truth annotations in our data

| Subject | | 1 | 2 | 3 | All |
|---|---|---|---|---|---|
| Puzzle | Length (s) | 150 | 120 | 120 | 390 |
| | Ground Truth coverage | 31.7% | 53.6% | 39.3% | 40.8% |
| Storytelling with toys | Length (s) | 640 | 540 | 615 | 1795 |
| | Ground Truth coverage | 6.9% | 20.1% | 14.4% | 13.4% |

## 3.2 Methodology

We report the performance of our pipeline on these videos with precision and recall metric scores. We show each component's added value for the pipeline with metrics to justify its need in the pipeline by running different versions of our pipeline that include different combinations of components. We name and describe them as follows:

1. P1: Each handstop candidate is considered a positive detection (Section 2.5.1)
2. P2: The candidates of P1 are filtered by candidate refinement (Section 2.5.2)
3. P3: Before and after image is retrieved. The remaining candidates are considered a positive detection (Section 2.5.3).
4. P4a: Filter the candidates of P3 by the image difference of $I_{t_1}$ and $I_{t_2}$ (Section 2.5.8)
5. P4b: Only apply long-term optical flow filtering described in (Section 2.5.6) and image difference (Section 2.5.8)
6. P4c: Only apply short-term optical flow filtering described in (Section 2.5.7) and image difference (Section 2.5.8)
7. P4d: Only apply body-occlusion filtering described (Section 2.5.5) and image difference (Section 2.5.8)
8. P4: Apply all inference removing filters and image difference utilizing the full pipeline.

P1 selects candidate time instances from the video and all other pipeline versions filter these initial candidates. Each pipeline's output is a list of candidate time instance that can be compared to the ground truth annotations with pattern recognition metrics, like precision and recall.

Since certain pairs of pipeline versions differ only in one algorithmic component (P1-P2, P2-P3, P3-P4a, P3-P4b, P3-P4c, P3-P4d, P3-P4) the difference in their performance shows the effect of adding said component to the pipeline. Each

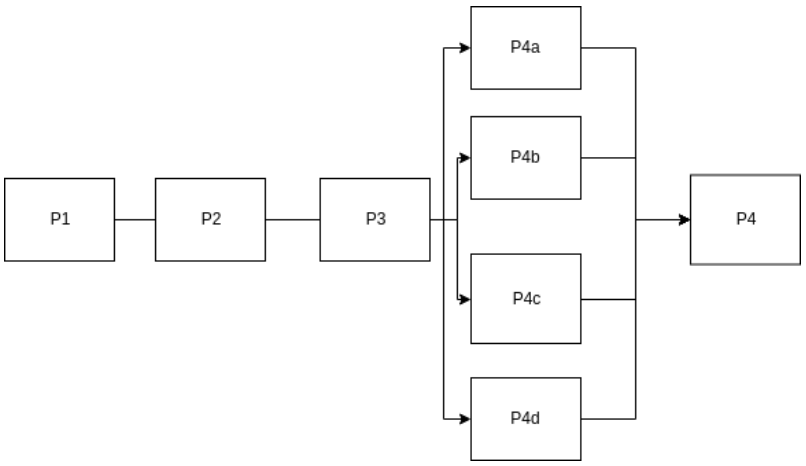



Figure 2: Overview of the stages described in Section 3.2. The stages until P3 are built linearly. Stages after this correspond to the different inference removal algorithms and are considered to be run in parallel. Finally the results are combined and image difference is taken in P4.

components filters the output of the previous one and we can treat it and evaluate it as a binary classification tasks.

The pipeline versions P1-P3 are built linearly by adding components, P4a-P4 are combinations of the 3 inference removal methods (Section 2.5.4).

## 3.3 Results

We report precision and recall scores for our evaluations (Table 2). Since taking an object happens over time, the ground truth annotations consist of intervals. We do not differentiate between left and right hand events, but we consider parallel annotations. On the other hand our method detects time instants.

We consider all positive predictions that fall into a ground truth interval to be true positives, and all ground truth interval that contain a positive prediction is handled as recalled.

P1 starts with a large number of candidate points measuring in good recall but poor precision scores. Progressing to P3, the candidates are filtered and we can see that precision increases and recall decreases. Comparing P4a-P4 we can see that the combinations of the 3 interference removal components has the best precision and precision and recall values vary when we remove only one type of inference (P4b-P4d).

Our full pipeline (P4) shows reasonable recognition capability when the dataset of the two activities are aggregated (precision 0.51, recall 0.49). When we consider the datasets separately, the "Puzzle" activity dataset has 0.7 precision with 0.68 recall, and "Storytelling with toys" has 0.41 precision with 0.41 recall.

Table 2: Precision and recall scores for each of pipeline variations for our datasets. The pipelines differ on which algorithmic components they contain, explained in Section 3.2.

| ADOS-2 activity | Metric | Evaluation pipelines | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | P1 | P2 | P3 | P4a | P4b | P4c | P4d | P4 |
| Puzzle | Precision | 0.45 | 0.62 | 0.60 | 0.61 | 0.65 | 0.61 | 0.68 | **0.70** |
| | Recall | 0.95 | 0.92 | 0.77 | 0.77 | 0.75 | 0.75 | 0.71 | **0.68** |
| Storytelling with toys | Precision | 0.16 | 0.19 | 0.21 | 0.22 | 0.26 | 0.22 | 0.38 | **0.41** |
| | Recall | 0.92 | 0.87 | 0.62 | 0.62 | 0.56 | 0.62 | 0.41 | **0.41** |
| All | Precision | 0.20 | 0.25 | 0.28 | 0.29 | 0.35 | 0.29 | 0.49 | **0.51** |
| | Recall | 0.93 | 0.89 | 0.67 | 0.67 | 0.62 | 0.67 | 0.51 | **0.49** |

(a) $I_{t_1}$ (b) $I_{t_2}$ (c) Finals mask applied to $I_{t_1}$

Figure 3: A semi-failure case from our evaluations. (a) and (b) shows the images selected before and after the hand stop time instant respectively. (c) shows the masked image after applying our filters. The object is correctly discovered, but also many other area are also segmented, most of them corresponding to shadow.

Furthermore, we carried out an analysis of the performance of each component by estimating their ability to correctly classify candidates as true or false positives in their respective place in the pipeline.

# 4 Discussion and outlook

For any '*detector*', low precision and high recall means that it is more general than intended, covering other events as well, while high precision and low recall means that it is too specific, covering only a subset of the targeted events. In the first case, one needs to integrate more knowledge into the pipeline. In the high precision case semi-supervised machine learning, where the annotated samples are collected with the help of the high precision detector on available non-annotated data, may become feasible. This made possible by tuning the individual deep neural network

components providing lower scores with the samples gained via consistence seeking [14].

We can see from the results that our pipeline's intermediate and final performance has higher metric values on the "Puzzle" dataset. We must take into account the ground truth to video length ratio (Table 1) for the two dataset. On "Puzzle" one has 40.8% chance with a random time instance to find a ground truth interval while the same chance is 13.4% for the other dataset. Thus the detection of our target events in the "Puzzle" scenario are easier.

We emphasize that adding each component increases precision, conclusively we are successful in adding knowledge to our pipeline by combining rules and trained deep neural networks. Our components still have shortcomings which we analyzed qualitatively. First, one of the limitations for our method is how we deal with shadows. See the example in Figure 3. This case is not covered by any of our filters at this point. Secondly, we found that there are multiple cases when the detected time instance of hand stop (interpreted as grabbing or releasing) is very close to a ground truth interval but outside of it. In future we consider representing the machine detections as intervals between the before-after images instead of time instances, and use IoU (Intersection-over-Union) measure for evaluations. This would make the detections easier to cluster or drop unlikely ones resulting in less false positives and less multiple detections of the same ground truth interval.

We also minimize the collection of new training samples as proposed in [14]. Our example is the monitoring of the manipulation of unknown objects and detecting "picking up" and "putting down" these objects. In turn the problem treated belongs to the family of '*zero-shot learning*' tasks. First, we considered the general driving principles of the process and transformed them into concrete rules by taking into account trained deep neural networks dealing with hand detection, body pose estimation and motion information extracted from optical flow estimation.

We carried out a detailed analysis of the proposed method on real world scenarios. We found that considerable complexity arises in this relatively simple problem and that it can be overcome by applying rules. We also note that information pieces, e.g., information about depth are missing and could be included. Due to novel developments in deep learning technologies, such as

1. the estimation of 3D distance of objects [10] and that of
2. 3D body configurations [15, 22], as well as the
3. precise estimation of hand configurations from monocamera recordings [20]

our approach on zero shot learning guided by neural-symbolic approach and the belonging self-training capabilities will become more precise and fit the stringent requirements of remote monitoring in the near future.

We plan to conduct a larger scale experiment in the future where we can address the mentioned issues and analyze the algorithm further on more data. We will also try to improve our pipeline by adding depth information estimated from RGB recordings using the listed deep learning algorithms.

## Acknowledgements

We are thankful to our supervisor Dr. habil. András Lőrincz and also to Zoltán Tősér for their great advices. Special thanks are due to Szilvia Szeier and Kevin Hartyáni for creating the ground truth annotations and to Judit Fülöp for verifying them. We would also like to thank Dr. Erzsébet Csuhaj Varjú as professional leader. The project has been supported by the European Union, co-financed by the European Social Fund EFOP-3.6.3-16-2017-00002.

## Author contributions

A.S. designed the theoretical background, M.Cs. designed the method, M.O. and A.S. helped to further improve upon it. A.S., M.Cs. and M.O. all took part in designing and executing the computational analyses. A.S. and M.Cs. wrote the manuscript.

## References

[1] Abdulla, Waleed. Mask r-cnn for object detection and instance segmentation on keras and tensorflow, 2017.

[2] Ahad, Md Atiqur Rahman. *Motion history images for action recognition and understanding*. Springer Science & Business Media, 2012.

[3] Alayrac, Jean-Baptiste, Sivic, Josef, Laptev, Ivan, and Lacoste-Julien, Simon. Joint discovery of object states and manipulation actions. *arXiv preprint arXiv:1702.02738*, 2, 2017.

[4] Bellman, Richard E. *Adaptive control processes: a guided tour*, volume 2045. Princeton university press, 2015.

[5] Bishop, Christopher. *Pattern Recognition and Machine Learning*. Springer-Verlag New York, 2006.

[6] Dai, Jifeng, Li, Yi, He, Kaiming, and Sun, Jian. R-fcn: Object detection via region-based fully convolutional networks. In *Advances in neural information processing systems*, pages 379–387, 2016.

[7] Du, Pan, Kibbe, Warren A, and Lin, Simon M. Improved peak detection in mass spectrum by incorporating continuous wavelet transform-based pattern matching. *Bioinformatics*, 22(17):2059–2065, 2006.

[8] Duan, Kun, Parikh, Devi, Crandall, David, and Grauman, Kristen. Discovering localized attributes for fine-grained recognition. In *Computer Vision and Pattern Recognition (CVPR), 2012 IEEE Conference on*, pages 3474–3481. IEEE, 2012.

[9] Everingham, M, Van Gool, L, Williams, C, Winn, J, and Zisserman, A. The pascal action classification taster competition. *International Journal of Computer Vision*, 88:303–338, 2011.

[10] Godard, Clément, Mac Aodha, Oisin, and Brostow, Gabriel J. Unsupervised monocular depth estimation with left-right consistency. In *CVPR*, volume 2, page 7, 2017.

[11] He, Kaiming, Gkioxari, Georgia, Dollár, Piotr, and Girshick, Ross. Mask r-cnn. In *Computer Vision (ICCV), 2017 IEEE International Conference on*, pages 2980–2988. IEEE, 2017.

[12] Ilg, Eddy, Mayer, Nikolaus, Saikia, Tonmoy, Keuper, Margret, Dosovitskiy, Alexey, and Brox, Thomas. Flownet 2.0: Evolution of optical flow estimation with deep networks. In *IEEE conference on computer vision and pattern recognition (CVPR)*, volume 2, page 6, 2017.

[13] İlsever, Murat and Ünsalan, Cem. Pixel-based change detection methods. In *Two-Dimensional Change Detection Methods*, pages 7–21. Springer, 2012.

[14] Lőrincz, A, Csákvári, Máté, Fóthi, Áron, Milacski, Z Ádám, Sárkány, András, and Tősér, Z. Towards reasoning based representations: Deep consistence seeking machine. *Cognitive Systems Research*, 47:92–108, 2018.

[15] Mehta, Dushyant, Sotnychenko, Oleksandr, Mueller, Franziska, Xu, Weipeng, Sridhar, Srinath, Pons-Moll, Gerard, and Theobalt, Christian. Single-shot multi-person 3d body pose estimation from monocular rgb input. *arXiv preprint arXiv:1712.03453*, 2017.

[16] Mittal, Arpit, Zisserman, Andrew, and Torr, Philip HS. Hand detection using multiple proposals. In *BMVC*, pages 1–11. Citeseer, 2011.

[17] Ren, Shaoqing, He, Kaiming, Girshick, Ross, and Sun, Jian. Faster r-cnn: Towards real-time object detection with region proposal networks. In *Advances in neural information processing systems*, pages 91–99, 2015.

[18] Rohrbach, Marcus, Rohrbach, Anna, Regneri, Michaela, Amin, Sikandar, Andriluka, Mykhaylo, Pinkal, Manfred, and Schiele, Bernt. Recognizing fine-grained and composite activities using hand-centric features and script data. *International Journal of Computer Vision*, pages 1–28, 2015. DOI: `10.1007/s11263-015-0851-8`.

[19] Simonyan, Karen and Zisserman, Andrew. Two-stream convolutional networks for action recognition in videos. In *Advances in neural information processing systems*, pages 568–576, 2014.

[20] Spurr, Adrian, Song, Jie, Park, Seonwook, and Hilliges, Otmar. Cross-modal deep variational hand pose estimation. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 89–98, 2018.

[21] Teo, Ching L, Yang, Yezhou, Daumé, Hal, Fermüller, Cornelia, and Aloimonos, Yiannis. Towards a watson that sees: Language-guided action recognition for robots. In *Robotics and Automation (ICRA), 2012 IEEE International Conference on*, pages 374–381. IEEE, 2012.

[22] Véges, Márton, Varga, Viktor, and Lőrincz, András. 3d human pose estimation with siamese equivariant embedding. *arXiv preprint arXiv:1809.07217*, 2018.

[23] Wei, Shih-En, Ramakrishna, Varun, Kanade, Takeo, and Sheikh, Yaser. Convolutional pose machines. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 4724–4732, 2016.