UNIVERSIDADE FEDERAL FLUMINENSE
INSTITUTO DE COMPUTAÇÃO

TROY COSTA KOHWALTER

AN INFRASTRUCTURE FOR GAMEPLAY GATHERING AND ANALYSIS WITH PROVENANCE

NITERÓI
2018

Troy Costa Kohwalter

AN INFRASTRUCTURE FOR GAMEPLAY GATHERING AND ANALYSIS WITH PROVENANCE

Thesis presented to the Computing Graduate program of the Universidade Federal Fluminense in fulfillment of the requirements for the Ph.D. degree. Topic area: Visual Computing.

Advisors: Prof. D.Sc. Esteban Walter Gonzalez Clua
Prof. D.Sc. Leonardo Gresta Paulino Murta

Niterói
2018

Ficha catalográfica automática - SDC/BEE

K79a Kohwalter, Troy Costa
An Infrastructure for Gameplay Gathering and Analysis with Provenance / Troy Costa Kohwalter ; Esteban Walter Gonzalez Clua, orientador ; Leonardo Gresta Paulino Murta, coorientador. Niterói, 2018.
171 p. : il.

Tese (doutorado)-Universidade Federal Fluminense, Niterói, 2018.

1. Videogame. 2. Telemetria. 3. Visualização de dados. 4. Produção intelectual. I. Título II. Clua,Esteban Walter Gonzalez , orientador. III. Murta, Leonardo Gresta Paulino, coorientador. IV. Universidade Federal Fluminense. Escola de Engenharia.

CDD -

Bibliotecária responsável: Fabiana Menezes Santos da Silva - CRB7/5274

TROY COSTA KOHWALTER

AN INFRASTRUCTURE FOR GAMEPLAY GATHERING AND ANALYSIS WITH PROVENANCE

Thesis presented to the Computing Graduate program of the Universidade Federal Fluminense in fulfillment of the requirements for the Ph.D. degree. Topic area: Visual Computing.

Approved in May 2018.

APPROVED BY

Prof. D.Sc. Esteban Walter Gonzalez Clua – Advisor
IC-UFF

Prof. D.Sc. Leonardo Gresta Paulino Murta – Co-Advisor
IC-UFF

Profª. D.Sc. Aline Marins Paes Carvalho
IC-UFF

Prof. D.Sc. Daniel Cardoso Moraes de Oliveira
IC-UFF

Prof. D.Sc. Bruno Feijó
PUC-Rio

Prof. D.Sc. Regina Maria Maciel Braga
UFJF

Niterói
2018

For Nerys, my faithful and beloved companion.

ACKNOWLEDGMENTS

I would like to thank my mother and brother for supporting my decisions.

I am grateful to my cousins, aunts, uncles, and grandparents for providing me wonderful moments.

This thesis would not have been possible without the help, patience, and counsels of my advisors, Esteban Clua and Leonardo Murta.

I want to thank my fellow graduate colleagues in the computer science department for promoting a stimulating and welcoming academic and social environment.

I want to thank Juliana Freire and her graduate students for receiving me at NYU during my Ph.D sandwich.

I would like to acknowledge the financial, academic, and technical support of the Universidade Federal Fluminense, CNPq, FAPERJ, and CAPES.

And lastly, my deepest appreciation goes to Nerys, to whom I dedicate this entire thesis.

RESUMO

O resultado de uma sessão de jogo é consequência de uma série de eventos, decisões e interações que são realizadas pelo jogador. Compreender e extrair dados destas sessões pode ser importante para análise da jogabilidade, para o entendimento do perfil do jogador e até mesmo para otimizar o modelo de negócios aplicado no jogo. Muitas ferramentas e técnicas têm sido desenvolvidas pela indústria de jogos a fim de registrar e extrair dados de uma sessão de jogo. Um método bem-sucedido é *Game Analytics*, que tem como objetivo compreender padrões de comportamento dos jogadores para melhorar a qualidade do jogo e a experiência do jogador. No entanto, os métodos atuais de *Game Analytics* não são suficientes para capturar as influências de causa e efeito, muitas vezes implícitas no jogo, que levaram ao resultado alcançado em uma sessão, permitindo entender e identificar elementos com mais profundidade. Em um trabalho anterior, propusemos uma nova abordagem baseada em proveniência para capturar estas relações causais, fornecendo a base necessária para utilizar informações de proveniência em análises de jogos. Este trabalho amplia nossa abordagem original, propondo e implementando um framework concreto para rastrear, gerenciar e visualizar dados de proveniência durante o jogo. Através deste trabalho, podemos exibir os dados de proveniência em um grafo interativo para análises exploratórias, permitindo que desenvolvedores e analistas entendam os eventos e os resultados obtidos. Também propomos técnicas de sumarização automática para reduzir os dados de proveniência sem perder informações agrupando eventos sequenciais semelhantes que, por si só, não foram suficientes para gerar mudanças significativa no jogo. Além disso, levamos a análise de proveniência para um novo nível, permitindo a análise de múltiplos gráficos de proveniência simultaneamente ao gerar um grafo de proveniência resumido para análise. Este gráfico resumido é útil para designers de jogos, podendo auxiliá-los na detecção de padrões de comportamentos de jogadores, identificar problemas não relatados pelos testadores, confirmar hipóteses formuladas pela equipe de desenvolvimento e até mesmo em questões de monetização do jogo.

Palavras-Chave: proveniência, grafo, jogos, métricas, telemetria, analises, sumarização, visualização, diff, merge.

ABSTRACT

The outcome of a game session is derived from a series of events, decisions, and interactions that are made during the game. Understanding and extracting data from these sessions is important to analyze the gameplay, to understanding the player's profile, and even to validate the business model applied in the game. Many tools and techniques have been developed by the game industry to track and store data from a gaming session. One successful method is game analytics, which aims at understanding the player behavior patterns to improve game quality and enhance the player experience. However, the current methods for analytics are not sufficient to capture the underlying cause-and-effect influences that shape the outcome of a game session and, therefore, allowing deeper understanding and interpretation of the game features. In our previous work, we proposed a novel approach based on provenance to track and record these causal relationships, providing the necessary groundwork to use provenance information in game analytics. This work extends our original approach by providing a concrete framework for tracking, managing and visualizing provenance data during the game. Through this work, we can plot the provenance data in an interactive graph for exploratory analysis, allowing developers and analysts to better understand the events and outcomes. We also propose automatic summarization techniques to reduce the provenance data without losing information by clustering similar sequential events that alone were not enough to generate any meaningful change in the game. Furthermore, we take the provenance analysis to a new level, allowing the analysis of multiple provenance graphs simultaneously by generating a summarized provenance graph. This summarized graph is useful for game designers, to aid in the detection of patterns in player's behaviors, to identify issues not reported by game testers, to confirm hypotheses formulated by the development team, and even testing monetization issues.

Keywords: provenance, graph, games, metrics, telemetry, analytics, summarization, visualization, diff, merge.

LIST OF FIGURES

LIST OF TABLES

LIST OF ACRONYMS AND ABBREVIATIONS

CG – Current Graph

CI – Confidence Interval

CMDS – Classical Multidimensional Scaling

GUI – Graphical User Interface

GVM – Gameplay Visualization Manifesto

HCI – Human-Computer Interaction

IDE – Integrated Development Environment

IPAW – International Provenance and Annotation Workshop

NPC – Non-Player Character

OPM – Open Provenance Model

PING – Provenance in Games

PINGU – Provenance in Games for Unity

QA – Quality Assurance

RPG – Role-Playing Game

SDDPG – Shortest Diff Distance to a Positive result Graph

SDM – Software Development Manager computer game

TRUE – Tracking Real-Time User Experience

TABLE OF CONTENTS

CHAPTER 1 – INTRODUCTION

1.1 CONTEXT

Methods for automatic telemetry of game sessions data have become an important component of game design and production in the last few years (EL-NASR; DRACHEN; CANOSSA, 2013). The collected data can be used for different purposes, such as behavior understanding and analysis (DRACHEN et al., 2012), bug detection (DRACHEN; CANOSSA, 2009a; GAGNÉ; SEIF EL-NASR; SHAW, 2012), balancing the game experience (PEDERSEN; TOGELIUS; YANNAKAKIS, 2010), users classification (DRACHEN et al., 2013), understanding common behaviors (WEBER et al., 2011), and even improving the monetization process (EL-NASR; DRACHEN; CANOSSA, 2013). In this sense, these collected data may help on increasing not only the game value but also the ARM (Acquisition, Retention, and Monetization), a valuable aspect of the nowadays digital entertainment business model. More recently, Machine Learning and Neural Network approaches are showing to be important methods for automatic game calibration and even runtime game content creation.

While the analysis of collected game data can be challenging due to the huge amount of generated information, visualization techniques (KEIM, 2002) can be used as a powerful solution for exploring and understanding game fluxes[1]. Furthermore, visualization techniques allow for faster exploration of the data and provide a higher confidence in findings from exploratory analysis of the data (KEIM, 2002). Moreover, it is also useful when the designer is not familiar with the gathered data or has vague analysis goals (LIU et al., 2011). Therefore, game log visualization methods are gaining popularity among designers for understanding gameplay[2] gathered information in the game industry (DRACHEN; SCHUBERT, 2013; WALLNER, 2013).

A common visualization method is the heat map (DRACHEN; CANOSSA, 2009b), which uses colors in a two-dimensional map to reflect the density of certain variables in particular locations of the game. Recently, *BioWare Inc.* used heat maps to analyze common bug locations in SWTOR (ZOELLER, 2010), while *Valve* used heat maps to analyze

[1] Game flux is the chronological order of all executed actions, events, and causal relationships that occurs during a game session.

[2] Gameplay is defined as “the total experience provided by a game’s structure and mechanics” (THOMPSON; BERBANK-GREEN; CUSWORTH, 2007).

multiplayer maps in Team Fortress 2 (AMBINDER, 2009). Meanwhile, *Bungie* and *Microsoft* used heat maps to determine common places where players died in *Halo 3* (ROMERO, 2008; THOMPSON, 2007). The *Game Analytics* tool (WULFF; HANSEN; THURAU, 2017) of the Unity game engine (HIGGINS, 2017) and Unreal® also allows for visualizing game data as heat maps directly on the scene, identifying bottlenecks and hotspots, showing underused and overused areas of the game. Another approach similar to heat maps was proposed by Nascimento (2010), which renders trail lines in the game map to represent paths taken by characters in the environment. However, current methods for tracking and visualizing data do not consider the causal relationships (*i.e.*, cause-and-effect) between the actions and events during a game session and need to be inferred by the game designer.

1.2 MOTIVATION

The aforementioned problems and challenges can be summarized into four main categories:

- Determining causal relationships between actions and events;
- Storing and displaying game session data;
- Information overload;
- Understanding the outcome.

The conclusion of a game session derives from a series of decisions and actions made throughout the game. During a game session, the player faces many challenges that require making decisions and actions in order to overcome them. Common practices of telemetry data include collecting state changes and event data; State changes lacks contextual information and provides only an overview of the session. Meanwhile, event data provides more detailed and fine-grained information about what transpired during the game session. However, neither of these practices used by the industry take into consideration the causal relationships that occurred during the session. These cause-and-effect relationships between the actions and events are an important factor for determining the reasons that led to a certain outcome.

Displaying game session data is also an issue in present times. An usual method to display game session data includes heat maps to show the density of data distributed throughout the different regions of the game. Other approaches use node-link diagrams (JUSTESEN et al., 2017; LIU et al., 2011; MAHLMANN; DRACHEN, 2016; SIFA et al., 2016; WALLNER, 2013) to represent states or the sequence of events. However, those approaches offer only basic visualization features for visual exploratory analysis with static views or limited interactable features to display the telemetry data.

Depending on the usage context, the amount of generated data can reach great sizes that affect negatively in the capability of analyzing it. Thus, the industry is always searching for more sampling strategies that can minimize the impact during analysis and new visualization methods that can scale with big data sets.

Succeeding or failing a game is the final consequence of a series of decisions, planning, and execution of a strategy to reach the final goal. Thus, an important problem is trying to understand the reasons that led to certain outcomes and why the user failed to achieve the final goal while another player managed to reach the desirable goals with a similar strategy. Currently, the game industry uses artisanal methods to understand the aspects that could have led players to fail certain goals by data crunching, using metrics or heat maps to find probable causes.

1.3 PROVENANCE

In a previous work, we introduced the use of provenance to improve the learning process in the context of serious games. Provenance is well understood in the context of art or digital libraries, where it respectively refers to the documented history of an art object, or the documentation of processes in a digital object's life cycle (PREMIS WORKING GROUP, 2005). The *Open Provenance Model* (OPM) (MOREAU et al., 2007) was created during the *Provenance Challenge* (MILES et al., 2010), which is a collocated event of IPAW. Shortly after, another provenance model was developed, named PROV (MOREAU; MISSIER, 2010), which can be viewed as the successor of OPM. Both models aim at bringing provenance concepts to digital data.

Provenance can be used for many purposes, from understanding how the data was collected in order to use it, to determine the object's ownership, and to decide if the information is trustworthy. Mainly, it is used to show the necessary steps to reproduce something by using an annotated causality graph, which is a directed acyclic graph enriched with annotations and is also known as provenance graph. According to Moreau *et al.* (2007), a provenance graph is the "*record of a past or current execution, and not a description of something that could happen in the future*". Similarly, the PROV model defines provenance as the "*information about entities, activities, and people involved in producing a piece of data which can be used to form assessments about its quality, reliability or trustworthiness*" (NIES et al., 2010).

Three different vertices in the provenance graph can represent the following provenance information in PROV: Agents, Activities, and Entities. Entities represent physical or digital

objects like a document, the Web, or material objects. *Activities* represent the actions taken in order to change or interact with *entities* or *agents*. Lastly, an *agent* is a person, software, organization, or *entities* that have responsibilities. Furthermore, several *agents* can have responsibilities over the same *activity* and a single *agent* can have responsibilities over several *activities*. *Agents* can also act on behalf of other *agents*, representing their interests when they are unavailable. These relations are some of the existing provenance relationships and are represented by edges in the provenance graph. Since the provenance graph captures causal dependencies between elements, it can be summarized by means of transitive rules.

Figure 1 illustrates an example of a provenance graph for the process of making a cake. Circles represent entities, squares are activities and the pentagons are agents. The entities in the graph, which are represented by circles, are the ingredients and materials used in the process. The agent is the cooker, which is responsible for executing the required activities to make the cake, which are mixing the ingredients, baking the cake batter and decorating the baked batter to produce the cake. The edges represent the causal relationships between the activities.

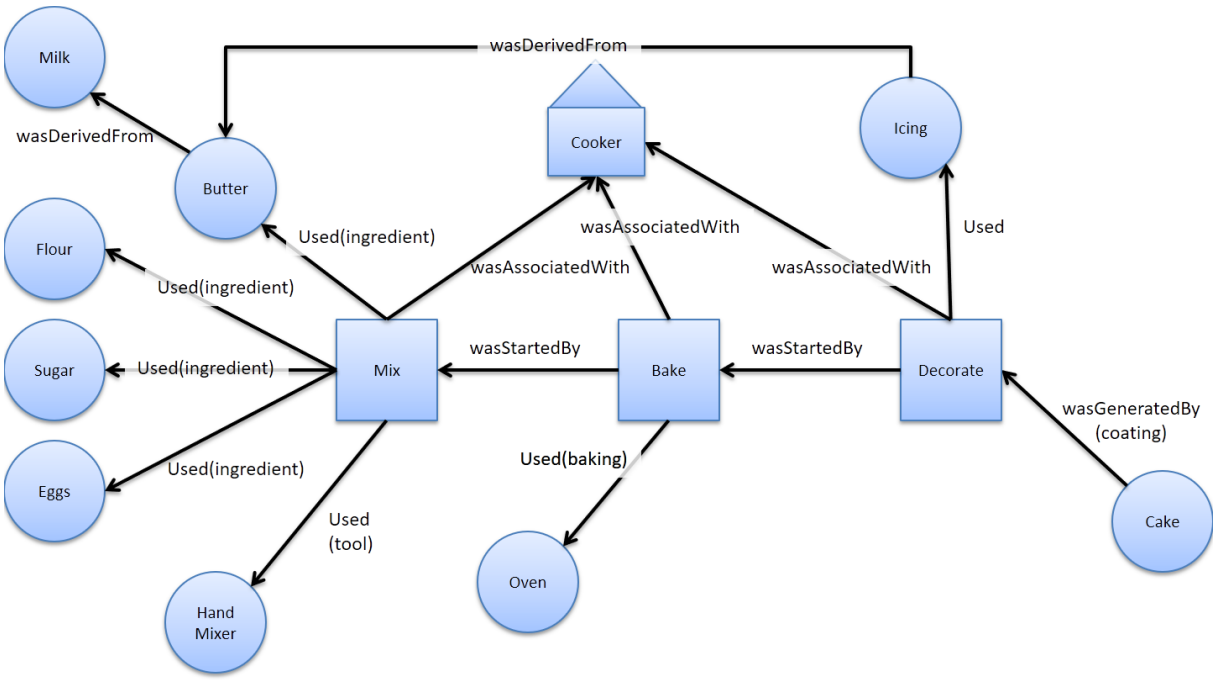


Figure 1: Provenance graph example for baking a cake

1.4 GOALS

Given the aforementioned motivation, the aim of this research is to improve the understanding of the game flux, providing insights on how the game session progressed and the influences on the outcomes. Therefore, our research hypothesis is:

"Does the use of provenance in the field of games make it possible to understand the objectives achieved during gaming sessions?"

In order to improve understanding of the obtained results from a game session, this work provides the means for tracking and storing the provenance data (DAVIDSON; FREIRE, 2008; MOREAU et al., 2007) while remaining game independent through a low-coupling solution for tracking game data and generating the provenance graph. Furthermore, we propose new provenance visualizations and new features for interactive graph manipulation for exploratory analysis. We also propose techniques for provenance summarization through similarity clustering to simplify provenance data. Lastly, we also propose a method for debugging that compares multiple provenance graphs to determine divergences that led to different outcomes.

As such, the main contributions of this thesis, considering the goal of supporting game designers during the analytics process, are:

1. *PinGU*[3]: A framework for tracking and managing provenance data, with a concrete implementation of a Unity based component (PinG for Unity);
2. *Prov Viewer*[4]: A provenance graph visualization tool for exploratory analysis;
3. Automatic provenance summarization through neighbor similarity, implemented in *Prov Viewer*;
4. *Prov-DIFF*: Provenance Graph Merge and Comparison, implemented in *Prov Viewer*.

1.5 RESEARCH METHODOLOGY

We defined five stages in this work in order to achieve our goals. The stages are: (1) problem characterization, (2) the conception and implementation of the concrete framework *PinGU*, (3) the conception and implementation of *Prov Viewer*, which is responsible for our provenance visualization techniques, (4) the conception and implementation for automatic provenance clustering, and (5) conception and implementation of a provenance merger and comparison method. Here, we present an overview of these stages:

Problem characterization: this stage started in a previous work where we detected the lack of tracking causal relationships in known game telemetry approaches after analyzing

[3] http://gems-uff.github.io/ping/

[4] http://gems-uff.github.io/prov-viewer/

popular works in the game telemetry and analytics field. With this in mind, our previous work (KOHWALTER; CLUA; MURTA, 2012) introduced the usage of digital provenance in games in order to detect these cause-and-effect relationships. The main goal of that work was to propose a conceptual framework, named *Provenance in Games* (PinG), which collects information during a game session and maps it to provenance terms, providing the means for a post-game analysis. As a proof of concept, we initially applied this conceptual framework to a serious game named SDM (KOHWALTER; CLUA; MURTA, 2011), which focuses on teaching Software Engineering processes. The provenance support in SDM allowed for a broader range of analysis by using collected game session provenance information to generate a provenance graph (KOHWALTER; CLUA; MURTA, 2013). Furthermore, we demonstrated in another work (KOHWALTER; CLUA; MURTA, 2014) that analyzing a game session with provenance provides better results, while also being faster, than analyzing the game without access to provenance data.

However, our *Provenance in Games* approach was still limited and in its earlier stages, providing only the idea along with the required groundwork in order to use provenance in Game Analytics. Furthermore, our initial visual analysis of the provenance graph was only through temporal information, ignoring spatial data when rendering the graph. This type of temporal visualization for the graph, while informative for a game like SDM that has a restricted space and lack spatial movement, might not be sufficient in a more generic set of game styles. Moreover, our initial structure for tracking and storing provenance information during a game session presented problems in scenarios with some games that had multiple characters due to the decentralized approach to store information. Therefore, it required an additional step to construct the graph from all tracked data.

We also performed an initial evaluation with volunteers to determine if the provenance data and its causal relationships provided a better understanding of the game events over the game replay. The positive results of that evaluation allowed further research and study of using provenance in the games domain. **Chapter 2** outlines the state of the art in Game Analytics and other relevant work. It presents existing usages of game session analysis, ranging from gameplay data logging to game analysis in the game industry. Meanwhile, **Chapter 3** presents our previous work in this field, which introduced a novel approach based on provenance concepts in Game Analytics to provide richer data tracking for analysis. This chapter also presents our earlier evaluation results that inspired in the continuation of this research field.

Conception and implementation of the *PinG* framework: the problem characterization stage was fundamental for the conception of our framework because we noticed a lack in the literature about tracking causal relationships from game sessions and the positive impact they can have when trying to understand events and the reasons behind outcomes. Thus, we conceived a framework based on our initial conceptual framework in the following steps: (1) create a data structure to manage and store provenance information, (2) create a method to track events, and (3) create a method to detect causal relationships (*i.e.*, cause-and-effect between actions and events). **Chapter 4** presents our proposal of a concrete framework for provenance tracking, which vastly improves the previous conceptual framework to a low coupling solution. We demonstrate our provenance tracking framework with its instantiation for the Unity game engine and demonstrate its integration into four different Unity games throughout the chapter.

Conception and implementation of *Prov Viewer*: tracking provenance data is only the first step. The richness of tracked provenance data requires the need to display the provenance data to be able to explore its wealth of knowledge about the game session and comprehend the relationships between actions and events. Thus, we took existing node-link representations for graphs and expanded it with numerous visualization features to allow designers to interact with the provenance graph to better understand the underlying causes. As a result, we created the provenance visualization tool *Prov Viewer*. **Chapter 5** presents our advances in provenance visualization, which is a fundamental step for exploratory analysis of game data. This chapter also describes in detail our provenance visualization tool with all the proposed advances for provenance graph exploration and visualization. We present three detailed visualization case studies from two existing games and another simple analysis related to urban data domain.

Conception and implementation for automatic provenance clustering: provenance data provides a vast wealth of data. The amount of tracked data can overwhelm the developer with the sheer detail of the events that transpired during a game session. Thus, we began exploring methods to reduce the overall provenance graph without losing the richness of information contained in the data. As such, we developed techniques to summarize the provenance graph by clustering sequence of events that alone did not contribute enough to the story. We evaluated our approach with another popular clustering algorithm through two experiments to answer the following research question: *RQ: Which type of similarity summarization is an effective method for reducing the information to be analyzed?* The results from both experiments showed that one of our proposed algorithms was the most effective similarity summarization approach. **Chapter 6** presents our work to deal with information

overload in provenance graphs. We describe our approach to summarize the provenance graph by clustering sequential events in the graph that alone did not contribute enough to cause a state change. We also provide two detailed evaluations of our approach to determine its effectiveness.

Conception and implementation of a provenance debugger: up to this point we are only focused on analyzing a single provenance graph at a time. However, we know the need to be able to see how multiple players fared in the game and compare each performance to understand why some paths lead to failure while others succeeded in their goals. Thus, we developed a provenance graph merger and comparison methods that allows combining multiple provenance graphs together into a single unified graph for analysis. The comparison allows the designer to detect sections of the graph that are different from other graphs, which can be used to find out the dissimilarities and the reasons behind each outcome. When used outside the games domain, these features can be used to debug experimental trials to determine why a specific trial failed while another had positive results. **Chapter 7** describes our approach to analyze and compare multiple provenance graphs. Our approach creates a unified provenance graph through graph merges and allows for provenance comparisons to find the divergences in the graphs that led to different outcomes. We also provide an evaluation to measure the accuracy of our comparison approach to detect the sections of the graph that might have led to negative outcomes.

1.6 PUBLICATIONS

The research developed during this thesis was published in conferences from Games and Software Engineering, including our initial proposal that originated during the Masters in 2014. We also published two journal articles. Our publications are:

1. KOHWALTER, Troy; CLUA, Esteban; MURTA, Leonardo. Reinforcing Software Engineering Learning through Provenance. *2014 Brazilian Symposium on Software Engineering (SBES)*, p. 131–140, set. 2014.
2. KOHWALTER, TROY C.; Oliveira Thiago; Freire Juliana; CLUA, ESTEBAN W. G.; MURTA, LEONARDO G. P. Prov Viewer: a graph-based visualization tool for interactive exploration of provenance data. In: *International Provenance and Annotation Workshop (IPAW)*, 2016, v. 9672. p. 71-82.
3. KOHWALTER, TROY C.; MURTA, LEONARDO G. P.; CLUA, ESTEBAN W. G. Capturing Game Telemetry with Provenance. In: *Brazilian Symposium on Computer*

Games and Digital Entertainment (SBGames), 2017. XVI Simpósio Brasileiro de Jogos e Entretenimento Digital, 2017. Best paper award in the computer track.

4. KOHWALTER, T. C.; MURTA, LEONARDO G. P.; CLUA, ESTEBAN W. G. Filtering Irrelevant Sequential Data out of Game Session Telemetry though Similarity Collapses. *Future Generation Computer Systems - The International Journal of eScience*, 2018.
5. KOHWALTER, T. C.; MURTA, LEONARDO G. P.; CLUA, ESTEBAN W. G. Understanding Game Sessions through Provenance. *Entertainment Computing*, 2018.

Furthermore, there were two more collaboration works based on our *Provenance in Games* approach:

1. JACOB, Lidson; KOHWALTER, Troy; CLUA, Esteban; DE OLIVEIRA, D.; MACHADO, A. A Non-intrusive Approach for 2D Platform Game Design Analysis Based on Provenance Data Extracted from Game Streaming. *2014 Brazilian Symposium on Computer Games and Digital Entertainment (SBGAMES)*, p. 41–50, nov. 2014.

2. JACOB, Lidson B.; KOHWALTER, Troy; MACHADO, Alex; CLUA, Esteban W. G. A Game Design Analytic System Based on Data Provenance. *Entertainment Computing – ICEC 2013*, Lecture Notes in Computer Science. p. 114–119, 2013. Accessed: 5 fev. 2015.

In addition to these works, we presented in the *ICSE 2017 Ph.D. and Young Researchers Warm Up Symposium* a preliminary proposal for using provenance data to better understand the game session.

CHAPTER 2 – GAME ANALYTICS

2.1 INTRODUCTION

During the earlier stages of the game industry, game development process lacked any pressing need for player data gathering and monitoring to better gain knowledge about users behavior (HULLETT et al., 2011). However, with the increasing popularity and demand for quality games and an increasingly aggressive market, the industry had to turn their eyes on finding ways to discover how their customers (players) are reacting to their games. This attention is focused both in a market point of view (sales) and user experiences, which is one of the most important objectives for game developers and is directly related to the player retention rate[5] (WEBER et al., 2011). Thus, the game industry began to adopt methods for gathering game data in order to understand better their clients. Since then, game data gathering gained attention during the game development stage, resulting in a widespread adoption of business intelligence practices for obtaining quantitative measures of user-oriented game data (EL-NASR; DRACHEN; CANOSSA, 2013). This change of paradigm of incorporating analytics has brought numerous benefits in the decision-making progress during the game development and became well known and accepted by the game developers.

As such, game analytics became an emerging field that is very popular and important for business intelligence in the game industry. However, it lacks standardization of key aspects and strategies. Nevertheless, game analytics provides a wealth of information for game designers, including feedback about design and gameplay mechanics, player experience, production performance, and even market reaction. Thus, the main goal of game analytics is to support the decision making the process at operational, tactical, and strategic levels for game development and is the main source of business intelligence for the game industry (EL-NASR; DRACHEN; CANOSSA, 2013).

From the many concepts that permeate game analytics, the three most common ones are telemetry, metrics, and data mining. The first concept is used to gather and store game data while the game is being played for future analysis. The second concept is used to map the raw telemetry data in measurable quantities for rudimentary analysis. Finally, the game data-mining

[5] "Player retention rate" refers to the percentage of people who played the game in month 1 and are still playing at month 2.

concept is used to find the wealth of hidden information inside the stored telemetry data, enabling more complex types of analyses.

Thus, this chapter presents the fundamental concepts for understanding the proposed method and concludes with known research approaches by the academia related to capturing game flux data. These related works can be classified in two categories: (1) logging strategies, to capture and record contextual information, and (2) visualizations strategies, for data mining over the logged game flux data, which represent state transitions during a game or execution of actions.

This chapter is organized as follows: Section 2.2 describes the benefits brought by game analytics, while sections 2.3, 2.4, and 2.5 present the concepts of game telemetry, game metrics, and game data mining, respectively, which are integral parts of game analytics. Section 2.6 presents popular research works that are related to game analytics and Section 2.7 provides a comparison between approaches. Lastly, Section 2.8 raises the final considerations of this chapter.

2.2 BENEFITS OF GAME ANALYTICS

Currently, game analytics are used for (1) understanding how the game community behavior with the game is, (2) improving gameplay, (3) detecting gameplay problems, and (4) predicting player behavior.

Initially, game designers used statistical techniques to gather player data in order to understand various aspects of their customers (*i.e.*, players). For example, DeRosa (2007) described how *BioWare* used statistics during play testing to determine where players spent their time and which special powers were used. Other researchers also tried to analyze movements during battles (HOOBLER; HUMPHREYS; AGRAWALA, 2004) and to discover how the players are spending their time in the game (DEROSA, 2007; DRACHEN et al., 2012; MOURA; EL-NASR; SHAW, 2011) and how they are playing it (DRACHEN; CANOSSA, 2009a). Game analytics is also used to understand how much time the community spend playing the game (WILLIAMS et al., 2009).

Another benefit of using game analytics is using tracked data to improve the gameplay aspects of the game by aiding during game validation and refinement (FULLERTON; SWAIN, 2008) and for developing better AI-controlled opponents or changing the game to adapt to player behavior (MISSURA; GÄRTNER, 2009; PEDERSEN; TOGELIUS; YANNAKAKIS, 2010; YANNAKAKIS; HALLAM, 2008). It is also useful for identifying gameplay elements

that are responsible for maintaining active players (WEBER et al., 2011, p. 11) and to develop user profiles that can be used by game developers to improve their games (DRACHEN et al., 2013, 2016; DRACHEN; CANOSSA, 2011; DRACHEN; CANOSSA; YANNAKAKIS, 2009). These type of information can also be used for marketing analysis, figuring out how to convert nonpaying players in a free-to-play game to paying customers (KING; CHEN, 2009).

Another usage for game analytics is detecting weak spots in the game design (DRACHEN; CANOSSA, 2009a; GAGNÉ; SEIF EL-NASR; SHAW, 2012; THOMPSON, 2007). Moreover, a similar usage is to discover illicit methods used by cheaters and gold farmers, as well as the people that are exploiting it, in an MMORPG game (MELLON, 2009; THAWONMAS; KASHIFUJI; CHEN, 2008).

Recently, game analytics also began to be used for predicting information about players, such as when they will stop playing the game (HADIJI et al., 2014; MAHLMANN et al., 2010; SIFA et al., 2016; XIE et al., 2014) and their in-game behavior and actions while playing the game (ESMAEILI; WOODS, 2016; WEBER; MATEAS, 2009). It is also being used for characterizing and predicting player behavior as the game evolves over time (MAHLMANN; DRACHEN, 2016; PIRKER et al., 2016; SIFA; BAUCKHAGE; DRACHEN, 2014), for mining behavioral data (BAUCKHAGE; DRACHEN; SIFA, 2015), and for social network analysis (LOSUP et al., 2014; PIRKER et al., 2018).

2.3 GAME TELEMETRY

Game telemetry is the automated communication process that collects data from remote clients (player's computer) and transmits this data to the game server (EL-NASR; DRACHEN; CANOSSA, 2013). This game data can be quantitative data about how players interact with the game. This tracked data, which is collected over the distance by game telemetry, can be related to game development, game research, remote monitoring and game server analysis, and player behavior.

In essence, game telemetry data is raw data about how the player interacts with the game that was derived remotely from a game execution. In general, these data represent attributes from game objects (*e.g.*, characters and items), such as the player's location in the map or damage values. This behavior data is normally collected by an embedded code component in the game that transmits the data to a specialized server for storage. However, there are other types of data that can be gathered by telemetry, such as data related to server load performance or network traffic.

Thus, Game telemetry is a very powerful tool to be used during game development, especially for tuning games, QA, testing, and finding elements in the game that need to be calibrated (ISBISTER; SCHAFFER, 2008). However, in order to make it useful and effective, it is necessary to correctly integrate it into the game and implement good strategies for data gathering. These strategies aim at minimizing the gathering of useless information that compromises the analysis process either by hindering it with too much information or by simply not recording unnecessary data.

2.4 GAME METRICS

With the advance of the game industry and its need to improve their games to better satisfy their customers (*i.e.*, players), the application of game telemetry became very popular during the development process. However, telemetry is only responsible for gathering game data. In order to analyze and understand the data, it is necessary to structure and describe the raw data through metrics. Metrics has been around for a long time and game metrics is derived from other fields, such as software engineering (EL-NASR; DRACHEN; CANOSSA, 2013). However, the increased need for telemetry, and consequently game metrics, expanded their application and its interest rapidly grew in the past few years in the game industry (EL-NASR; DRACHEN; CANOSSA, 2013).

Similar to other applications of metrics, game metrics are measures of data derived from gameplay information, such as quantitative measures of attributes from game objects. The most common source of game metrics comes from player behaviors, which is recorded by game telemetry as raw data. This raw data gathered from telemetry is transformed into measurable quantities, such as total playtime, death-ratio, daily active users in the game, and a number of kills during a game session. However, it is not limited to the player behavior in the game. Game metrics can also be used to measure the number of sales for a particular game, how many game items or units were sold on the first week, and the number of complaints made either by players or by employees in the last month. It can also be more related to the software development point of view, such as task completion rates and server or client hardware performance. These are all considered game metrics, even if they are related to Business Intelligent (RUD, 2009) because they all share a common ground: they relate to some aspect of games, including the development process. Currently, game metrics are categorized into three different types (MELLON, 2009): *Player*, *Performance*, and *Process* metrics.

The *Player* metrics measures what players are doing throughout the game session. This type of metric is responsible for collecting data related to player statistics, such as the frequency of items used by players, their attribute points allocation, the total time taken to finish a quest, and which quests they prefer to play, among many other possibilities. Overall, this metric is used to make the game more fun or for improving gameplay.

While Player metrics are concerned with data related to how users play the game, the *Performance* metrics are related to technical and software infrastructure behind the game. The Quality Assurance (QA) teams heavily use this metric in order to monitor the game's processes to ensure quality, such as server stability and the frame rate on the game client. Furthermore, this metric is derived from traditional software engineering performance and QA techniques.

The last type is the *Process* metrics, which focus on the process of developing games, monitoring, and managing the development process. This metric is also based on common strategies used outside the games domain, such as those present in software engineering.

2.5 GAME DATA MINING

The increasing need by the game industry for game analytics during development, and later for monitoring online games, has the side effect of increasing the amount of gathered game data, as well as data's complexity. This gathered data, which is collected during a game, provides a wealth of detailed information about player behavior from multiple players. However, this dataset increase requires efficient analysis methods that must be capable of scaling with the dataset size, while still providing easy and readable data results for game developers. This field of research is called Game Data Mining, which deals with the challenges of providing to developers and designers insights from game data collected via game telemetry (EL-NASR; DRACHEN; CANOSSA, 2013).

The wealth of hidden information in the game telemetry is enormous. However, they can be difficult to discover without proper data mining, an expert professional analysis for the game, or adequate data visualization. This led to tracking the majority of game data by game telemetry and only logging and storing it, waiting for proper analysis methods (EL-NASR; DRACHEN; CANOSSA, 2013). Thus the challenge the game industry faces when analyzing telemetry data can be associated with the general challenge of working with *Big Data* (EL-NASR; DRACHEN; CANOSSA, 2013). Simply gathering information and populating databases is not enough to help game developers in their decision-making process.

Therefore, deciding how to employ the game telemetry in a game is not a trivial task (KIM et al., 2008b) since it is necessary to decide which data to record, how it can be used for analysis, and how it will be transformed from its original raw form to actual knowledge that is useful for the developers. Recently, new methods have been proposed to assist the game data analysts and decision makers to find hidden gems inside the logged game data (BAUCKHAGE; DRACHEN; SIFA, 2015), predicting player retention rate (SIFA et al., 2016) and outcomes (MAHLMANN; DRACHEN, 2016), supporting making better decisions and fine-tuning their games (EL-NASR; DRACHEN; CANOSSA, 2013), and monetization aspects (SIFA et al., 2015). Furthermore, the neural networks, deep learning, and machine learning in general are a recent tendency in games for analysis (PARK et al., 2017), knowledge discovery (BROWN et al., 2017), and to imitate humans (JUSTESEN et al., 2017). These methods are based on Visual Data Mining and have become a promising solution for exploring and understanding game fluxes. The following subsection explains the concept of Visual Data Mining, followed by known approaches that use this type of analysis.

2.5.1 **VISUAL DATA MINING**

Visual Data Mining presents the tracked game data in some form of visual representation to allow developers and designers to quickly scan through a huge amount of tracked data visually in order to extract knowledge, gain insights, draw conclusions, and directly interact with the data. Visual Data Mining is also useful when the designer is not familiar with the gathered data or has vague analysis goals (LIU et al., 2011). This type of analysis is based on the premise that a person can recognize patterns better than a computer algorithm in graphical data. Therefore, Visual Data Mining is usually used for exploratory analysis of the data. It can be seen as a hypothesis generation process and is gaining popularity among designers for understanding gameplay data (WALLNER, 2013).

Figure 2 illustrates a common example of Visual Data Mining through heat maps in the *Unreal Engine* to show the locations on the map with kill events by using a specific weapon. This visualization gives the idea of map coverage by using colors in the heat map that range from purple (almost no activity) to red (highest activity), allowing designers to quickly identify the "hot spots" in their game. The following subsection details more types of analytics that are based on spatial information and uses the Visual Data Mining approach.

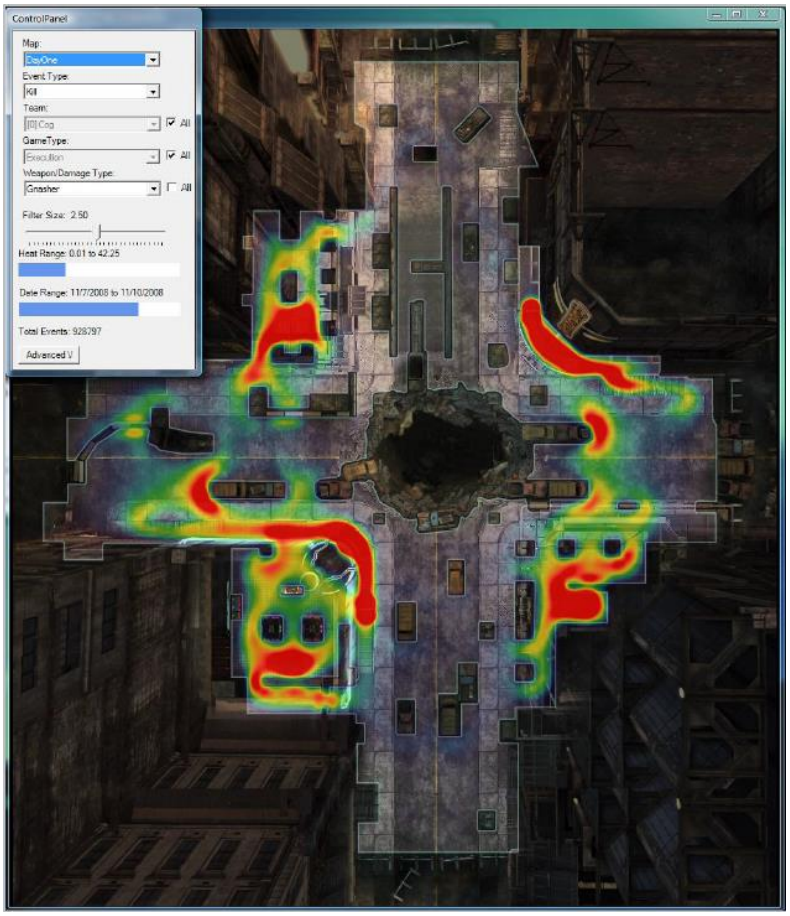


Figure 2: A heat map representing shotgun kills on Gears of Wars 2. Source: Schoenblum (2010)

2.5.2 **SPATIAL DATA ANALYSIS**

Spatial data analysis is based on the Visual Data Mining approach. It aims at discovering knowledge by extracting patterns through visual representations of the tracked data, taking into consideration spatial and temporal information. This type of analytics is important because temporal aspects of player behavior are vital for game progression analysis (DRACHEN; CANOSSA; YANNAKAKIS, 2009). However, detecting patterns considering spatial and temporal data is often more complex, requiring specialized methods to explicitly consider spatial and temporal information in addition to others more commonly information, such as player attributes. Due to its increasing importance in game analytics, Drachen and Schubert (2013) presented a survey of relevant works on spatial analysis, classifying them in four groups: *univariate* and *bivariate* analysis, *multivariate* analysis, trajectory analysis, and *behavioral* analysis.

The first group, which is *univariate* and *bivariate* visualizations, represents the most common techniques used by the game industry and academia (DEMERS, 2008; DRACHEN;

CANOSSA, 2011; HOOBLER; HUMPHREYS; AGRAWALA, 2004; KIM et al., 2008b). These types of visualizations are based on the evaluation of one or two variables from the telemetry data, providing an easy to read and intuitive interpretation of the data distribution. Examples of this type of spatial analysis include the aforementioned heat map and resource distribution in a game map. Developers internally use this type of information to evaluate designs. However, the player community can also use them for guidance.

The second group, defined as *multivariate*, enables developers to evaluate how multiple attributes or variables interact in the game (DRACHEN; CANOSSA, 2011). This analysis can also be achieved by combining two or more visualizations of the first group. The main goal of this type of analysis is to visualize the relationship between multiple features in the same region of the game. For example, analyzing locations where players died combined with weapons kills, filtering these locations by a specific weapon. Another example is combining the player orientation with death events, revealing where the player was facing when he was killed.

The third group is the *trajectory* analysis, which describes the movements of game objects during a specific period of time (BAUCKHAGE et al., 2014; MILLER; CROWCROFT, 2009; PAO; CHEN; CHANG, 2010), such as the player navigation process through a game level. This type of analysis is useful for understanding the player's actual game experience, examining group behavior, studying tactics adopted by players, and locating illegal actions or behaviors, such as using a program to automate player actions in online games.

The fourth group is the *behavioral* analysis, which represents various methods used to identify behavior patterns or to classify players into groups or profiles. This type of analysis tries to identify patterns by examining the successive sequence of actions that are executed in order to achieve a certain goal. More importantly, this analysis also considers the context when these actions were executed because different situations can lead to different types of behavior. For example, a wounded player that has low health will try to get a health restoration item while another player with full health will tend to skip it and save for a later moment when he might need.

2.6 STRUCTURED LOGGING AND ANALYSIS TECHNIQUES

This section describes in more detail some well-known approaches proposed by the academia and the game industry that are related to game session analysis, outlining techniques for game telemetry and data visualizations. The criterion used for selecting the approaches is similar to snowballing sample (GOODMAN, 1961). The sampling procedure starts with a finite

individual population as seed. Each seed in the sample is asked to name different individuals in the population. These newly named individuals, who form the second stage, are asked to name more individuals, forming the third stage. This process runs recursively until no new individual is named.

For the seed, we used Play-Graph (WALLNER, 2013), a recent research that displays gameplay data by using graphs. From the seed, we obtained another graph based visualization (WALLNER; KRIGLSTEIN, 2012) and an approach that combines behavioral and contextual data visualization (KIM et al., 2008b). From these, we selected another approach for understanding player behavior (DIXIT; YOUNGBLOOD, 2008), however we do not describe it in this chapter because it focuses only on visual representations of movement behavior (*e.g.*, walking, jumping, and pirouettes) to overcome obstacles in the environment and ignores interactions and all other types of game data, which is the opposite of what we want. Nevertheless, from that approach, we selected the last one, which proposes a framework for gameplay data logging (JOSLIN; BROWN; DRENNAN, 2007).

2.6.1 **GAMEPLAY VISUALIZATION MANIFESTO**

Joslin (2007) proposed the *Gameplay Visualization Manifesto* (GVM), which is a framework for gameplay data logging that uncovers gameplay events by attaching logging methods in game objects responsible for generating relevant events during the game. The logging method gathers information according to an event model, describing which attributes and information are logged for each event type. For example, interaction events log information related to the identification of the objects in the interaction.

The event model is the basis for a game data logging framework. It encapsulates the information that is desired by users and classifies the events into three groups: immersion, quest, and social. The immersion group represents events related to increasing the player's sensation of being involved in the game flux. The quest group represents events related to quest creation, execution, and analysis. Lastly, the social group represents events related to social factors in the game, such as group meeting or interaction with other characters.

Aside from classifying events in groups, they are also categorized into three different types: time events, interaction events, and emergent events. The time type represents events that are logged at predefined time intervals. Interaction type represents events related to interactions with other game objects and is parameterized by frequency, such as logging the attack event every third strike. The emergent type represents events that occur from internal

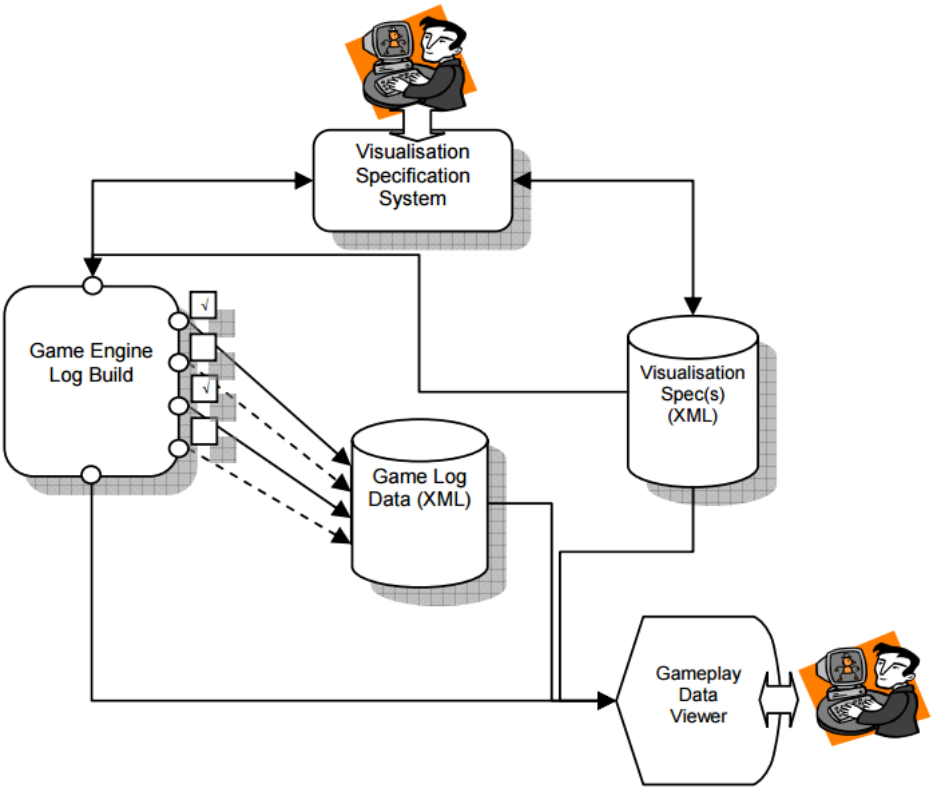


Figure 3: GVM logging architecture. Source: Joslin *et al.* (2007).

state changes, such as dying from loss of health. With this event classification, the information to be logged can be customized for each group and type. Figure 3 illustrates GVM's framework for logging data, showing the specification system, which configures the data to be captured, and the gameplay data viewer, which allows designers to analyze the game log data collected during gameplay.

The data logging framework has four different log specification interfaces: In-situ, Aggregate, Programmer, and Player. The in-situ interface is an in-game interface to display the event log, allowing designers to log events via the game interface as they occur. The aggregate interface summaries logged information by using graphics to facilitate the tracking of data-streams. The programmer interface allows the programmer to specify data logging from the source code inside the programming IDE. The programmer interface is mainly used to log game data for debugging purposes. Lastly, the player interface allows for players to provide feedback on various aspects of the game by reporting their experience. This interface is integrated with the game and periodically asks questions to the testers about their impressions on several aspects of the game, adding subjective impressions of fun experienced by the player in the log for future analysis.

This data-logging framework was mainly designed for online games in order to increase efficiency in gameplay verification process, reduce testing expenses, and improve quality assurance. The framework logs game data by using event models designed for logging, while also providing some basic data-stream visualization.

However, the main application is for collecting game metrics, such as player deaths, position, time spent in available certain actions (*e.g.*, crafting and fighting), item usage (*e.g.*, equipment), actions performed, and player enjoyment. Therefore, GVM does not track cause-and-effect relationships. It tracks only the executed actions along with their timestamp and location, in addition to character attributes and equipment.

Even though it tracks spatial information (*i.e.*, coordinates), it is only used to filter data by regions or localized sectors in the game (*e.g.*, show executed actions in the forest region of the map). There is no graphical representation of tracked data in the actual game scene. Thus, designers do not have an overview of the entire scene and are required to query specific sections to verify if there is some useful information. Furthermore, their log visualization is similar to spreadsheets with lines representing actions executed by each player, respecting their chronological order of execution. However, this approach does not combine these "lines" in order to create a unified visualization to extract patterns or offer an overview of the general behavior of players. Moreover, GVM has no technique to aid in the visualization and analysis of tracked data, especially when dealing with huge amounts of data. Due to these problems, GVM appears to be designed for a small-scale deployment, such as during the play testing phase of development.

2.6.2 **T.R.U.E.**

The *Tracking Real-Time User Experience* (TRUE) approach (KIM et al., 2008b) is another logging approach that combines human-computer interaction (HCI) instrumentation, which collects *user initiated events*[6] (UIEs) and logs file analysis techniques in order to automatically record user interactions with games. While the focus of HCI instrumentation is to collect the frequency count of events, TRUE logs the sequences of events along with their timestamps. These sequences of events are important in order to understand user behavior. While typical HCI instrumentation logs how many times the user accessed the Help function

[6] According to KIM (2008b), UIEs are "*events that occurred when the user interacted with the system*".

from the game, TRUE logs the previous sequence of events that led the user to use the Help function for each occasion.

Another key aspect of the TRUE approach is the type of collected information. Instead of recording generic low-level events, such as mouse coordinates and function calls, TRUE collects event sets containing both the event as well as contextual information from the event. For example, in a game where the player died, TRUE records the player's death, the equipment the player was using, the game's difficulty setting set by the player, the enemy that killed the player (if that was the case), and other useful elements that might determine the cause of the player's death.

TRUE architecture is illustrated in Figure 4. The data capture occurs at runtime, while the user is playing the game. The data capture collects events and their contextual information, along with timestamps indicating when they occurred. At the same time, TRUE captures digital videos of user's screen, which show her interaction with the game. The video is automatically synchronized with the event's timestamps and indexed, allowing jumping to particular events, relevant to the analysis. This link between event and video was stated by Kim (2008b) to be an effective approach for understanding the users' behaviors and how they interact with the game. The last data capture from TRUE is in the form of a survey available to the user after finishing her interaction with the game. The survey is aimed at capturing information that might have been missed by the tracked UIEs.

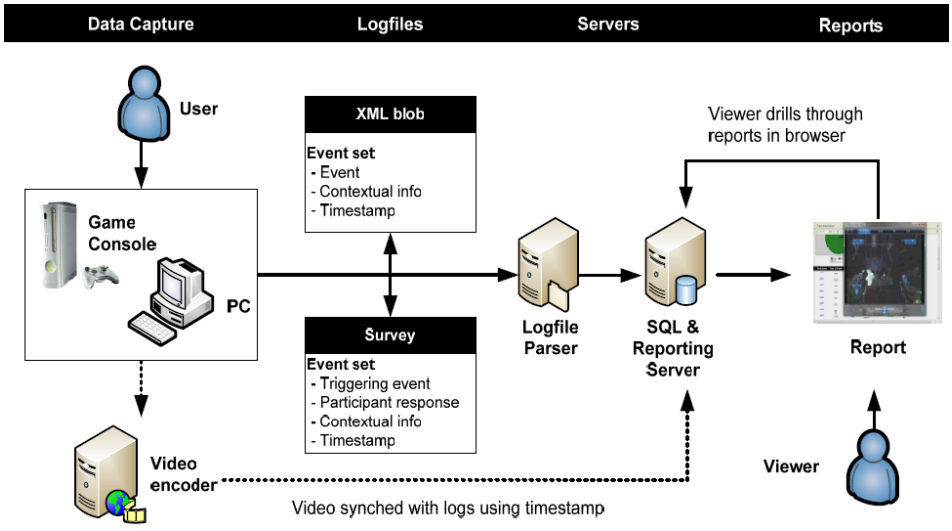


Figure 4: The TRUE architecture. Source: KIM *et al.* (2008b).

For example, when testing a game using the TRUE approach, a brief survey is displayed to the player asking whether she enjoyed the game and how difficult it was. This type of survey is used to avoid making wrong inferences about the game. For example, failing in a game

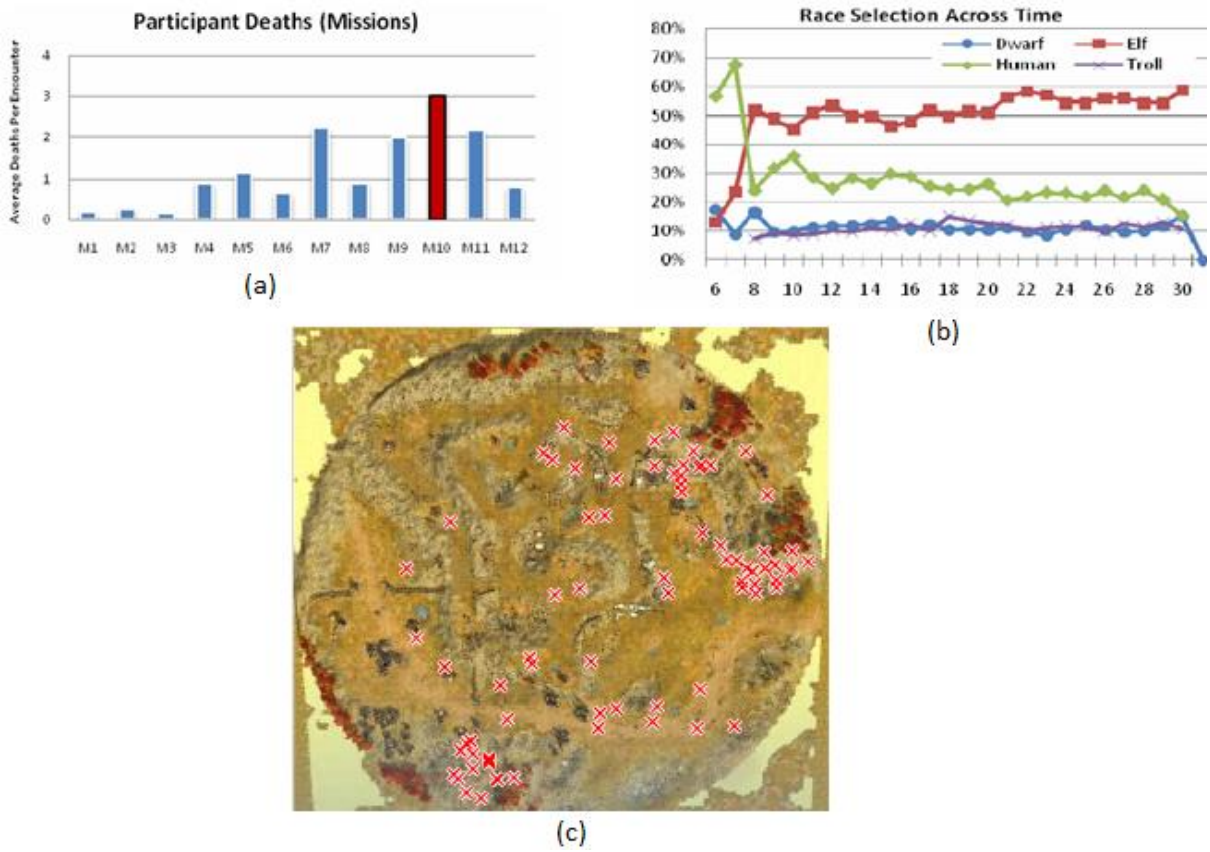


Figure 5: Data visualization from TRUE. Source: KIM et al. (2008b).

session may sometimes be a motivating part of the fun, while winning at the first attempt might indicate the game was too easy for the player.

The captured data is available to the analyst for analysis using visual representations in order to be easier to spot points of interest. The data visualization varies with the type of analysis. Figure 5 shows different examples of data visualization, customized by designers for the application. In the figure, there is a graph showing the average times of player death for each mission in a game (a) and which race players selected the most across time (b). Another possible visualization is by using an in-game map to display death locations in a Real Time Strategy game (c).

The TRUE approach is focused on the videogame industry and is designed to detect issues and understand the causes in the same way a usability test does. It also incorporates attitudinal behavior by using surveys to aid in the understanding of the player's emotional experience. Most of the tracked data is displayed by metrics (*e.g.*, death rates, item usage) through statistical graphics and representations.

The TRUE approach has another type of data visualization, which is based on its ability to tracks precise coordinates in the game world when an action or event occurs. However, the visual representation of the data, which is based on the tracked spatial information, is through points on the map. This visualization is very similar to a heat map but without grouping the information by density. In fact, TRUE does not offer any type of information clustering when dealing with big data sets besides the aforementioned statistical information. Furthermore, the spatial visualization illustrates only facts, without showing any additional information related to what caused these reported facts. For example, it displays death locations on a map but does not inform who or what killed the person in the same visualization.

Even though TRUE also track contextual information along with the event or action, the designer still needs to infer the reasons behind that led to an outcome. This occurs because this contextual information is nothing more than extra attributes that were tracked during the execution of the action (*e.g.*, equipment used) and not actual relationships between events. In fact, this inference mostly occurs due to the video recorded in the game session. Thus, when the designers find indicators that represent issues, he digs through the tracked data and watches the video to try to understand the issue and form recommendations to solve the problem.

2.6.3 **PLAYTRACER**

Playtracer (ANDERSEN et al., 2010; LIU et al., 2011) is a visual tool designed to illustrate how groups of players move through the game space. *Playtracer* can be used for behavior analysis in games with the concept of state transitions. The transitions in the game are represented as different states by applying Classical Multidimensional Scaling (CMDS) (COX; COX, 2010) to project the game space in two dimensions. Thus, *Playtracer* aids the designer by showing common pathways and alternatives that players used to succeed or fail in their tasks, identifying pitfalls and anomalies in the scene. It also tracks how players progressed through the levels of the game.

In this system, a play trace is a path that each player took in the game, scaled to two dimensions by using CMDS. The transformation places similar states close to each other while dissimilar states are placed apart. Thus, CMDS allows for easy similarity identification between states that were visited by players. The distance between states is calculated by following specific metrics that are customized by the game. Distance metrics are also used to analyze different features of the game. For example, a distance metric with a component to compare how many steps are necessary to reach a goal state will cluster goal states while placing states

that are difficult to reach the goal far away. Thus, the designer can identify players that are not making progress in their goals and possibly investigate the issue.

The input for *Playtracer* is a list of all states that the players visited during the game and a distance metric to calculate the distance between states. The output is an oriented graph where the vertices represent the game states and the directed edges are the movements the player did to change from one state to another. Furthermore, the size of the vertex is proportional to the number of players that reached that state. Thus, the size of the vertex can be used to identify which states were more visited by players.

Moreover, the graph utilizes color to distinguish displayed information. A yellow state is the game's initial state and a green state represents the goal. Blue edges represent movements made by players who won the game and red edges are for those who lost. The shades between red and blue represent the probability for the player who reached the state to successfully complete the game. Lastly, cycles in the graph represent failed attempts from the players, where they made a move that returned to a previous state.

As can be observed in Figure 6 most players moved from the initial state in yellow, at the center of the figure, to a purple state where most of the difficulties began, leading to movements to several different states further away from their goals. The figure also shows that the goal state, in green, could only be accessed by two different game states.

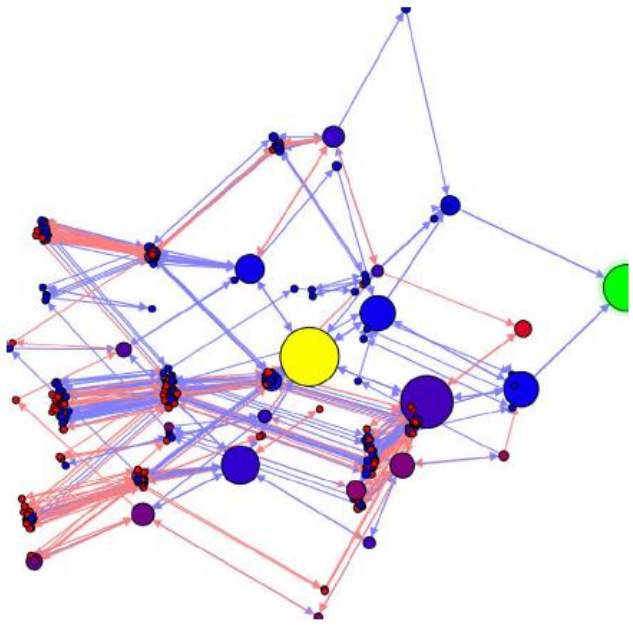


Figure 6: *Playtracer* state visualization. Source: LIU *et al.* (2011).

The main focus of the *Playtracer* is to display aggregated user behavior in a graph in order to aid in understanding common strategies adopted by players and to identify points of

confusion. To solve problems related to the game with many states, *Playtracer* uses features to aggressively cluster states together to make a cleaner visualization. Another feature is to make equivalent states to be represented by the same state, reducing the number of states displayed on the screen. Lastly, it is possible to filter the graph (*e.g.*, winners from losers) to visually compare their respective behaviors in order to identify similarities.

A drawback is that *Playtracer* does not take into consideration temporal information, thus they cluster similar states despite the fact that they could have occurred at very different moments of the game. The temporal information would allow stating the order of events in the game, shedding more light on the player's behavior. For example, *Playtracer* does not say when each state was visited by each player or the order they were visited, but only that they visited it while playing the game. Moreover, *Playtracer* does not use spatial information for analysis, disregarding the graph visualization in an actual map or a visual representation of the game. Thus, game designers need to guess or interpret the information in order to infer where it actually happed in the scene. Furthermore, *Playtracer* only states the facts (*i.e.*, game states) that occurred in the game sessions without providing information that might have caused the change of state besides the player's movement.

2.6.4 **PLAY-GRAPH**

Another graph-based approach is the *Play-Graph* (WALLNER, 2013; WALLNER; KRIGLSTEIN, 2012), which is a recent concept to formally describe and visualize gameplay data by using different graph visualizations to illustrate multiple variables and their interrelations along with the temporal progression of players. The gameplay analysis of *Play-Graph* illustrates the sequence of states visited by players based on their actions over the course of the game. In the *Play-Graph* context, a game state describes a certain configuration of the game or an entity, while actions consist on player interactions within the game, such as shooting, jumping, or using an object. These actions are responsible for changing the current game state due to influences generated in the current state.

In this concept, a game is viewed as a finite state machine with a finite number of states and transitions between them. Thus, the state machine can be represented by an oriented graph with each vertex representing a state from the game and edges representing actions. States are composed of a set of attributes from the game. Actions are triggered by players at a specific point in the game and can be of different types or have a specific duration. For example, possible types of actions are: running, walking, jumping, and interacting with an object. Furthermore,

actions (edges) linked to states (vertices) can have labels to provide additional information to differentiate from other states and actions.

The *Play-Graph* visualization is composed of node-link diagrams. Nodes, or vertices, in the graph represent game states. The size of each node is directly related to the number of players that visited that state at any time during the game. Moreover, multiple edges from the same source to the same target are merged together to create a meta-edge. The thickness of each meta-edge is proportional to the number of edges that composes the meta-edge. It is possible to have two meta-edges between two nodes due to the nature of the directed graph, where each meta-edge represents a different direction. Furthermore, each node and edge type in the graph is distinguishable by colors. The colors are chosen by the user in order to adapt the visualization to the user's needs. Lastly, icons in the graph represent players in the game. The icon color is directly related to specific attributes of the player (gender, age, character class).

Figure 7 illustrates the basic representations from the graph, showing player transitions from one state to another. Basic elements from the graph include nodes (a), which represent states, directed edges (b) representing player's actions, meta-edges (c), and the player icon (d), representing the time-dependent location of individual players. The red meta-edge near the center of the graph is composed of two edges, "mirror" (blue edge) and "rotate" (red edge), as can be seen by the window with the blue and red bars. This window also details the meta-edge

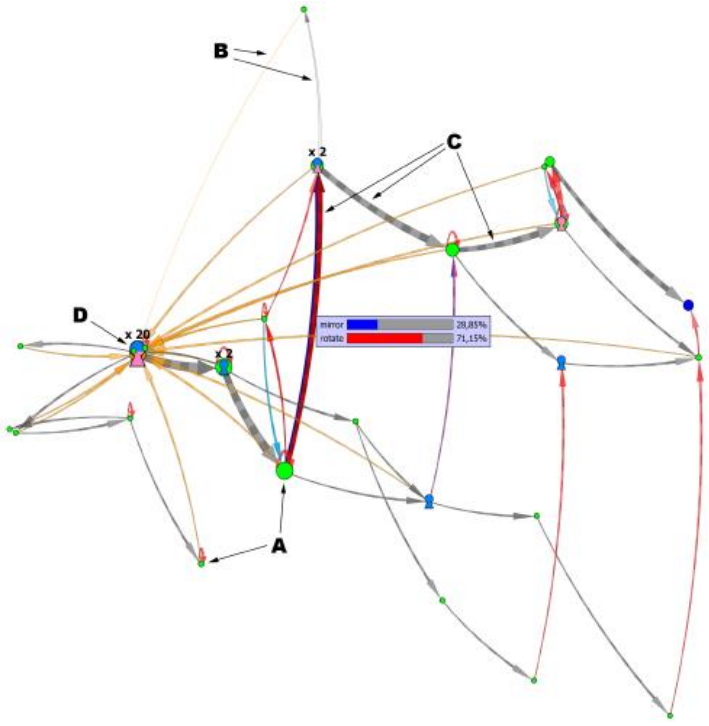


Figure 7: Basic elements from the *Play-Graph*. Source: WALLNER (2013).

composition ratio: 28.25% of the meta-edge's thickness is due to the number of "mirror" edges in it and 71.15% is from the edge "rotate". Meanwhile, Figure 8 illustrates another graph visualization from *Play-Graph* that uses spatial information to position state nodes in a game map.

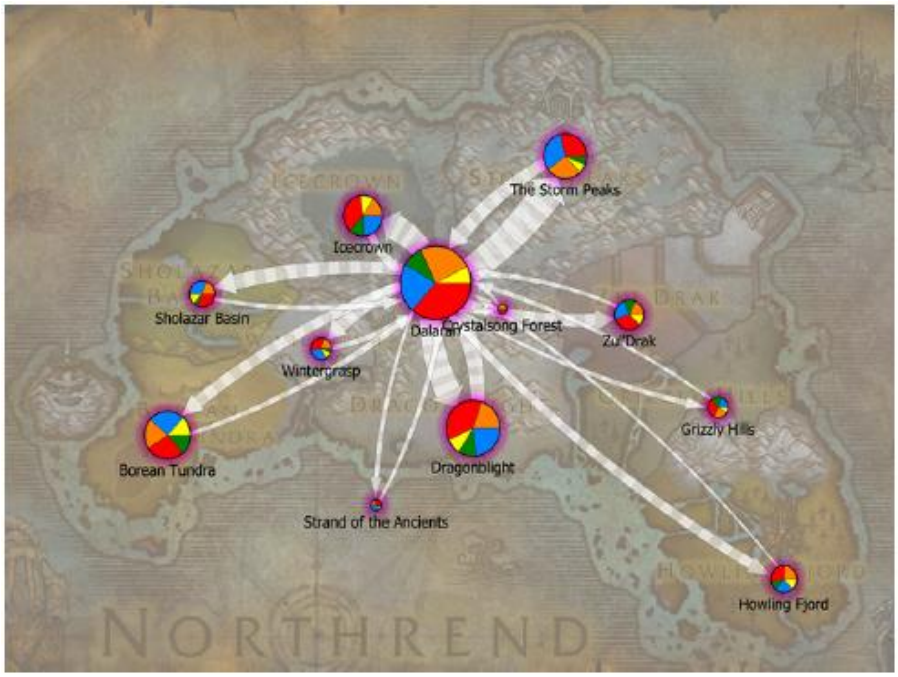


Figure 8: *Play-Graph* spatial visualization. Source: WALLNER (2013).

Viewing gameplay data by graphs allows the usage of graph theory concepts to study and understand player behaviors. The displayed graph from *Play-Graph* illustrates the player's progression in the game, showing actions the player made that change states. Unlike *Playtracer*, this approach uses temporal information to distinguish states, allowing for observations related to time-dependable events. However, temporal information is only used for player icons and not the nodes representing states. *Play-Graph* also allows comparing two different graphs from the game context, such as two versions of the same game, one before and after adding new features. This comparison support highlight areas where the player activity has increased or decreased with the new features. Furthermore, *Play-Graph* can also use spatial information to render the graph over the game scene and cluster nearby nodes to form a single node that provides statistical information in the scene's region (*e.g.*, player race distribution at that location), as illustrated by the colored nodes in Figure 8. However, the clustering algorithm disregards temporal information, since this information is not in the node (only in player icons), and indiscriminately clusters similar states that are spatially close. The many colors in the node from Figure 8 represent the percentage distribution of each node type that forms the cluster.

This node clustering, as well as edge clustering (represented by the edge thickness), is aimed at reducing the amount of information displayed in the graph, besides providing a statistical distribution view.

Moreover, due to the nature of how the graph is structured in *Play-Graph* (*i.e.*, limiting vertices as states and edges as actions), the understanding of player behavior is guided by the player progression in the game (*e.g.*, killed a boss), and not by how the player interacted with the world (*e.g.*, combat rounds from the battle against the boss). From the available documentation, there is no way to determine interactions or influences. Only the changes from one state to another, caused by an action executed by the player, can be identified. However, influences in the player's action, such as an influence from another character that affected the transition of one state to another, are not present in the graph (there are no edges linking edges). Besides, this visualization was not designed to closely track individual progression, but to track the player population flow.

2.7 COMPARISON BETWEEN APPROACHES

This section compares the described approaches from Section 2.6 by outlining their features according to the following characteristics:

- **Graph**: Indicates if the approach explicitly uses a graph to represent information.
- **Graphics**: Indicates if the approach explicitly uses graphical charts to represent information.
- **Event Context**: Indicates if the approach gathers contextual information from events and actions that can be used to improve the understanding of the event/action.
- **Actions**: Indicates if the approach collects details about the actions performed in the game, instead of only collecting the action's outcomes.
- **Statistical data analysis**: Indicates if the information gathered and/or displayed by the approach is used for statistical analysis (gameplay metrics).
- **Cause-and-Effect**: Indicates if the approach gathers information about factors that affected the outcome.
- **Temporal Information**: Indicates if the approach has any form of temporal analysis of the tracked data.

- **Spatial Information**: Indicates if the approach has any form of spatial analysis of the tracked data.
- **Information Overload Mitigation**: Indicates if the approach has any strategy to minimize information overload when providing the data for analysis.
- **Multi-session analysis**: Indicates if the approach has any built-in method for analyzing multiple sessions or data from different players at the same time.

Table 1 provides a comparison chart for the presented approaches. Fields filled with "√" indicate the approach supports the specified characteristic. Fields with "□" indicate the approach generates sufficient information that can be used for the specified characteristic. However, they do not explicitly use or treat the information. Fields filled with "**x**" indicate that the approach does not support, does not have the feature, or has no available documentation about the feature.

Table 1: Comparative chart of approaches

Features	GVM (JOSLIN; BROWN; DRENNAN, 2007)	TRUE (KIM et al., 2008b)	Playtracer (ANDERSEN et al., 2010)	Play-Graph (WALLNER, 2013)
Graph	x	x	√	√
Graphic	√	√	x	x
Event Context	x	√	x	x
Actions	x	√	x	√
Statistical Data Analysis	√	√	□	□
Cause-Effect	x	x	x	x
Temporal Information	√	√	x	√
Spatial Information	x	√	x	√
Information Overload	x	x	√	√
Multi-session analysis	x	x	√	√

The approaches presented in this Chapter are aimed at analyzing player behavior, game balancing, and game testing by identifying gameplay issues. Furthermore, they use graphs or graphics to illustrate the general population behavior, which is used by designers to improve the game. However, the approaches that track temporal and spatial information provide limited visualizations (if any), making game designers guess where and when the events happened in

the game by digging through tracked data. Furthermore, the clustering techniques used to deal with large amounts of data compromises, even more, these visualizations by losing the sequence of events due to the aggressive information compression. Moreover, they do not capture cause and effect information. They only capture the executed action, losing the causal relationships and factors that have had influenced during the execution of an action. Even TRUE, which captures contextual information, is not able to track factors that originated from the execution of other actions. It only tracks additional information from the character that executed it (*e.g.*, equipment used). Thus, they provide limited analysis possibilities, requiring more complex methods to extract knowledge from telemetry data. In other words, they still demand a considerable effort from the game designer to infer the possible reasons that led to an outcome.

2.8 FINAL CONSIDERATIONS

This chapter presented the field of game analytics, describing existing approaches from the game industry and the research community for game telemetry, metrics, and data mining. Furthermore, we presented related work that focuses on logging methods (telemetry) and gameplay analysis (data mining). However, there are other works outside the games domain that are related to our proposed approach. These will be discussed in their respective chapters.

Therefore, in the next chapter, we describe the basis of our proposal, which is the *Provenance in Games* concept, developed in a previous work to address those main problems. The chapter presents the initial conceptual framework and prototypes used to support this research, including early methods for tracking telemetry data and basic visualizations. It also presents initial results that motivated us to continue investing this research branch.

CHAPTER 3 – PROVENANCE IN GAMES

3.1 INTRODUCTION

As previously presented in Chapter 2, many games use game analytics to support the decision-making process at an operational, tactical, and strategic levels for game development. The tracked telemetry game data is crucial for game designers, allowing them to perform various types of analysis in order to better support their decisions. However, the tracked data provides a limited understanding of how events unfolded during the game and the way they affected the outcome. This is due to the fact that existing telemetry techniques do not gather causal relationships among actions and events.

Therefore, in our previous research (KOHWALTER; CLUA; MURTA, 2013), we developed a conceptual framework named PinG for tracking and managing provenance data. The original PinG conceptual framework provides the means to track provenance information during game sessions for rich post-game analysis based on cause-and-effect relationships. The provenance analysis is done by tracking gameplay data and generating a provenance graph, which relates the actions and events that occurred during the game session. The generated provenance graph is a visual representation of the tracked provenance data, allowing game designers to visually analyze and understand cause-and-effect relationships between events that occurred during a game session. Thus, they can identify critical actions that influenced the game outcome. We decided to use PinG as the basis of our proposal and extend it in order to reach all our aforementioned goals due to this ability to track cause-and-effect relationships.

This chapter briefly explains the core concepts of the previously developed conceptual framework named PinG (Provenance in Games) that records the *game provenance*, which was used as a basis for our current proposal. The *game provenance* is defined as the record of all the decisions, interactions and events executed throughout the game session, as well as their relationships and impacts over the course of the game. Furthermore, we describe the initial prototypes and findings from an earlier evaluation (KOHWALTER; CLUA; MURTA, 2014) that supports the usage of the conceptual framework for better understanding a game session. These promising results prompted us to further investigate this novel research branch of using provenance telemetry data in games. Everything discussed in this chapter was done during the master's program, while everything after this chapter is novel and accomplished during the Ph.D.

This chapter is organized as follows: Section 3.2 describes the conceptual framework previously developed in an earlier work. Section 3.3 presents the first usage of PinG in an existing serious game and the first prototypes for provenance tracking and visualization based on the conceptual framework. Section 3.4 provides the results of a previous evaluation of the prototypes, providing the motivation for this research. Finally, Section 3.5 provides the final considerations of this chapter.

3.2 CONCEPTUAL FRAMEWORK: PING

The *Provenance in Games* (PinG) conceptual framework (KOHWALTER; CLUA; MURTA, 2012) was developed to map provenance concepts to the context of games. PinG is based on the PROV model (GIL; MILES, 2010), which provides the basis for specifying information that was involved in creating or influencing a particular object. Thus, in order to use PROV, PinG provides a mapping of elements from the provenance domain to their corresponding elements in a game domain, relating each type of vertex of the provenance graph into typical elements found in games. It is important to note that the edges' orientation in the provenance graph goes from the present to the past, instead of the common orientation used in graphs, which are from the past to the future. Therefore, it is first necessary to define the provenance's counterparts in the game context in order to use the provenance graph to represent a game session.

A typical digital game architecture is mainly composed of game objects and the game loop. All objects present in a game, from environment objects to characters, are inherently defined as game objects. Game objects by themselves do not add characteristics to the game. Instead, they are containers that hold components that implement actual functionality, such as scripts (*i.e.*, artificial intelligence, player controller, *etc*.), meshes (the object structure or “body”), physics, textures, animations, and audio. Meanwhile, the game loop is responsible for the sequence of events that occur in a game, allowing the game to keep running regardless of the user’s input. The game loop keeps the game alive, updating game object states and executing their actions and behaviors. Each script in a game object has a function *update*, which is called by the game loop in order to execute the specific game object functionalities. Every time the game loop is ticked, it executes the *update* function of the scripts that belong to the game objects present in the scene.

The PinG conceptual framework provides the mapping of elements from the provenance domain to their corresponding elements in a game domain, mapping each type of vertex from

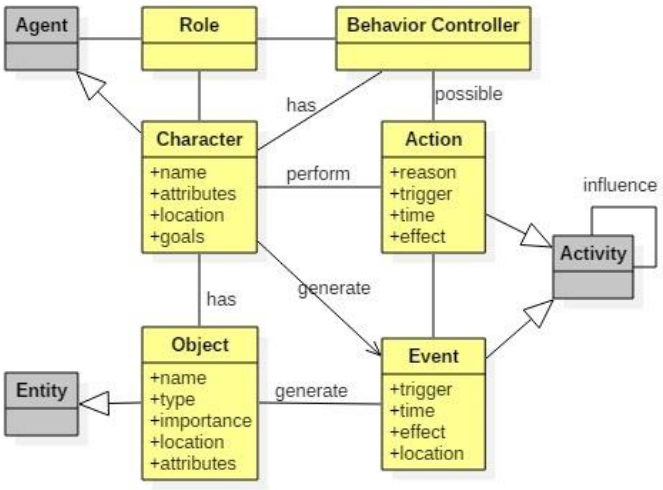


Figure 9: PinG data model diagram. Gray classes represent generic provenance classes.

the PROV model into elements that can be represented in games. In the context of provenance, *entities* are defined as physical or digital objects. Trivially, in the PinG approach, they are mapped into game objects without autonomous behavior, such as elements from the scenario. In provenance, an *agent* corresponds to a person, an organization, or anything with responsibilities. In the game context, agents are mapped into characters present in the game or game objects with autonomous behavior, such as event controllers, plot triggers, or the game's artificial intelligence overseer that manages the plot. Therefore, *agents* represent elements capable of making decisions or that have responsibilities in the game, while *entities* represent objects with no autonomous behavior. Lastly, *activities* are defined as actions taken by *agents* or interactions with *entities*. In the game context, *activities* are defined as actions or events executed throughout the game, such as attacking, dodging, and jumping. Figure 9 illustrates this mapping.

The PROV model defines relationships that can be used to express the context, such as the association relationship between activities and agents (*wasAssociatedWith*) or the influence an activity had on another activity (*wasInfluencedBy*). Furthermore, it also provides rules to extend these relationships or to create new ones. For instance, it is possible to create relationships to express that one character was spawned (created) by another character (*wasSpawnedBy*), such as a boss spawning minion. In addition, the PROV model deals well with the time flow due to represent the data in a causality graph in a timeline fashion, which can be heavily explored in games. Thus, these edges in the provenance graph, which represent causal relationships, should also express the type (*i.e.*, semantics) of the relationship between vertices.

Each character in the game should have an explicit model about its behavior in order to generate and control its actions. With this explicit model, a behavior controller can register information about the action when it is executed. These executed actions can be represented by a series of attributes that provide a description and context of the action, allowing the creation of a provenance graph. As illustrated by Figure 9, every action needs a set of information: a reason for its existence, why the action was performed, what triggered it (if applicable), and who performed the action. In addition, the time of its occurrence can be important depending on the reason of using provenance. The main reason of using provenance is to produce a graph containing details that can be tracked to determine why something occurred the way it did. Therefore, with this assumption, the time of the action, the agent who did it, and the effects of the action need to be recorded for future analysis.

Events also work in a similar way as actions, with the difference in who or what triggered them, since events are not necessary tied to characters. For objects, its name, type, location, importance, and the events that are generated by it can also be stored to aid in the construction of the graph. Lastly, agents can have their names, attributes, goals, and current location recorded. The provenance information collected during the game is used for the generation of the provenance graph, which can be used as a visualization tool for exploratory analysis. In other words, all the information collected throughout the game session is organized in the form of a directed graph. Thus, all relevant data should be registered, preferentially at a fine grain. The way of measuring relevance varies from game to game, but ideally, it is any information deemed relevant by the game designer that can be used to aid the analysis process.

3.3 PING CASE STUDY

In various games, some facts might not be clear or transparent enough for the player to understand why something went wrong. In some traditional games this can be solved with a new game session. However, sometimes the player or even developers want to have the opportunity to understand what the causes in an analytical way were. Thus, in a previous work (KOHWALTER; CLUA; MURTA, 2012), we proposed a novel approach using provenance in the game field.

The provenance analysis infrastructure, which uses the framework presented in (KOHWALTER; CLUA; MURTA, 2012), was instantiated in a software engineering educational game named SDM (Software Development Manager) (KOHWALTER; CLUA; MURTA, 2011). The goal of SDM is to allow for undergraduate students to understand the

existing cause-effect relationships in the software development process. Thus, the adoption of provenance has the potential to better support knowledge acquisition, allowing tracking mistakes made during a game session or identifying concepts that are not well understood by the students.

In SDM the player manages a team of employees that develop software according to contracts made with customers. The gameplay and game mechanics are modeled presenting possibilities to the player to decide strategies for development and define the roles and tasks for each staff member. As in any contract, the software has requirements that must be followed during development. From a gameplay point of view, these requirements help to balance the mechanics and rules. When the software is completed and delivered to the customer, there is a quality assessment of the software that modifies the project completion payment. Since SDM focuses on people management, the main elements of the game are the employees, which represent the player's labor force. Employees can perform different roles (analyst, architect, manager, marketing, programmer, and tester), which use the employees' human attributes to calculate their performance depending on the respective roles. Another attribute present in the game is specialization, which is used to define the employee working competence. With the specialization system, it is possible for employees to undergo training to learn new sets of skills. Also, the concepts of working hours, morale, and stamina are used to modify the employee's productivity.

Figure 10 illustrates the mapping of provenance concepts into the game context, outlining important information of each element type to be collected during game execution for provenance analysis. Yellow classes belong to the game domain, showed in Figure 9. Furthermore, these previously mentioned characteristics are illustrated in Figure 10 by the blue classes, which shows a simplified version of SDM's class diagram focusing on the employee. Each employee is defined by his human attributes (adaptability, auto didacticism, human relations, logical reasoning, meticulousness, negotiation, objectivity, organization, and patience), can have specializations categorized in three different types (with a total of 14 different specializations), and can be allocated for training in order to acquire new specializations.

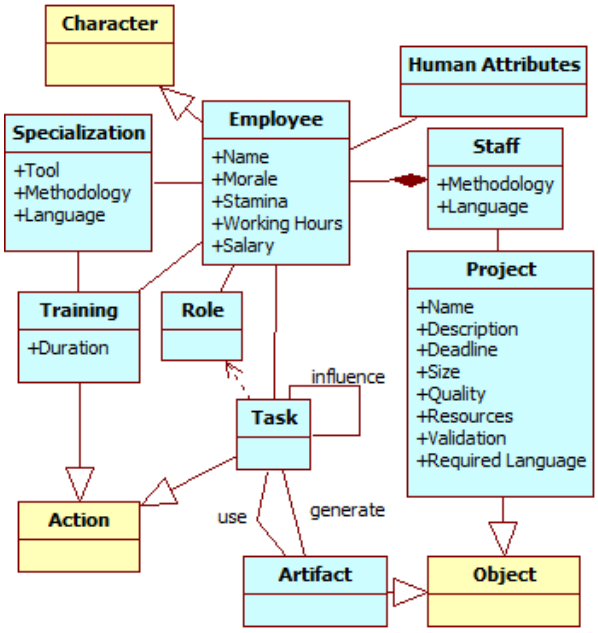


Figure 10: Mapping of the SDM game

Each employee can have up to two different roles at the same time, among six possible roles available. Each role has a different set of tasks, which are administered by decisions trees (MORET, 1982) that considers internal influences (attributes, morale, and stamina) and external influences (player or staff) to determine how these tasks are executed. Tasks can influence and be influenced by other tasks from another employee and can also generate artifacts, which can represent prototypes, used to validate software requirements or test cases (unit, integration, system, and user acceptance). Lastly, employees belong to the player's staff and develop the software for a customer, respecting the customer's requirements and deadlines.

3.3.1 **PROVENANCE TRACKING PROTOTYPE**

The data structure used in SDM to collect provenance information was adapted and mapped to be suitable for the proposed conceptual framework presented in (KOHWALTER; CLUA; MURTA, 2012), which is as follows: each project contains a list of employees involved in its development. In turn, each employee has a list of his actions executed throughout the development. If any action had an external influence during its execution, then the action also has a pointer to the action that influenced it. Throughout the game, the information about actions that are executed or triggered is collected at runtime and stored for later usage. Executed actions go to their respective employee lists. When new employees are added to the project, they receive their own list of actions and are added to the project's employee list. Each day of the game universe stores the state of the software development at the end of that day.

Since the information collected is used for the generation of the provenance graph, its context needs to be applied to one of the three possible types of provenance vertex: *activities*, *agents*, or *entities*. From the data model explained in (KOHWALTER; CLUA; MURTA, 2012) and previously mentioned at the beginning of this section: *activities* represent actions or events, *entities* represent static game objects (prototypes, test cases, software development state), and *agents* represent dynamic game objects (employees and clients).

The majority of the provenance gathering, which is related to *activities*, is administrated by decisions trees and occurs at leaf nodes of the tree, where actions are executed. The information gathered varies according to the element type, as can be seen in Figure 11. *Activities*' provenance information (c) is taken directly from the decision tree, getting the execution information and retracing the tree path from the leaf to the root. *Agents*' information (b) is gathered when they first interact in the game. *Entities*' information gathering varies according to the entity type. For example, the project as a whole (a) has its information gathered on a daily basis, recording the current state of development. On the other hand, prototypes and test cases *entities* have their provenance collected when they are created.

Moreover, the causal relationship between elements is also gathered. This occurs, for instance, when an *activity* is influenced by another *activity* or generates an influence on an *entity*. Examples of influences include an employee aiding another employee or when a task changes the state of the software under development.

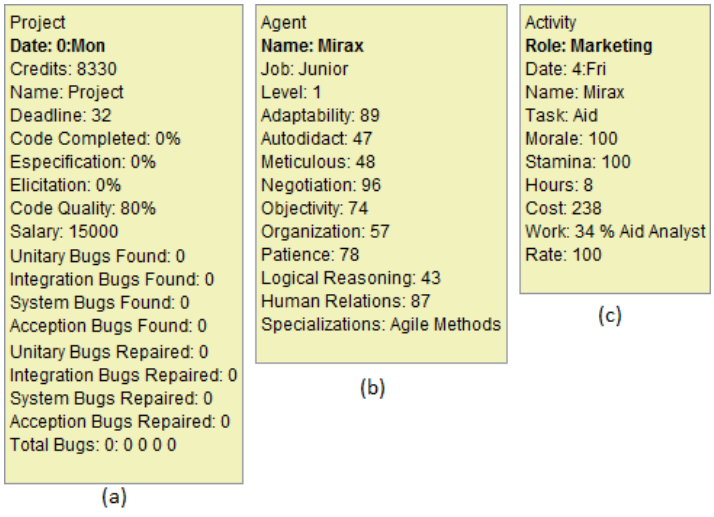


Figure 11: Provenance information regarding the project as a whole (a), an employee (b), and an action (c).

3.3.2 **PROVENANCE VISUALIZATION PROTOTYPE**

With the adaptations for provenance gathering made in the original SDM (KOHWALTER; CLUA; MURTA, 2012), it became possible to use the collected provenance data to generate a provenance graph for analysis. The collected game data is exported to *Prov Viewer* (KOHWALTER; CLUA; MURTA, 2013), which was a prototype tool for provenance graph visualization made specifically for SDM. In *Prov Viewer*, the game provenance data is processed and automatically used to generate an interactive provenance graph of the game session to aid the analysis process.

Figure 12 illustrates the graphical user interface (GUI) of *Prov Viewer* and the displayed provenance graph from a gameplay session generated by SDM. Using the visual notations defined in (MOREAU et al., 2007), a square vertex represents an *activity*, while circle represents an *entity* and an octagon represents an *agent*. The provenance graph is displayed at the center of the screen but only part of it is visible due to the graph size. However, it is possible to zoom in or out and navigate through the graph. The graph layout is set to be similar to a spreadsheet, where each "line" represents the *activities* of each *agent* and each "column" represents a day in the game. The filters, specifically defined for SDM, are located in the lower region of the interface. The "Collapse Agent" button collapses all the *agent's* vertices into the *agent* itself. It is useful to detect if an *agent* had any influence throughout the game, instead of looking vertex by vertex. The "Collapse" button allows the user to collapse selected vertices, creating a meta-vertex that summarizes edges (influences) by type. The "Extend" button removes the last collapse made to generate the selected meta-vertex.

The "Display Edge" is an important aspect of analysis, allowing for the identification of types of influences in the graph, filtering the graph edges that are not relevant for the desired analysis. The displayed graph only shows the selected edges types, omitting unselected types. For example, in Figure 12 the edge types "Neutral" and "Aid" are selected, thus showing all positive (green) and negative (red) influences of the "Aid" type and all "Neutral" (dotted-black) type edges, which in this case are association edges.

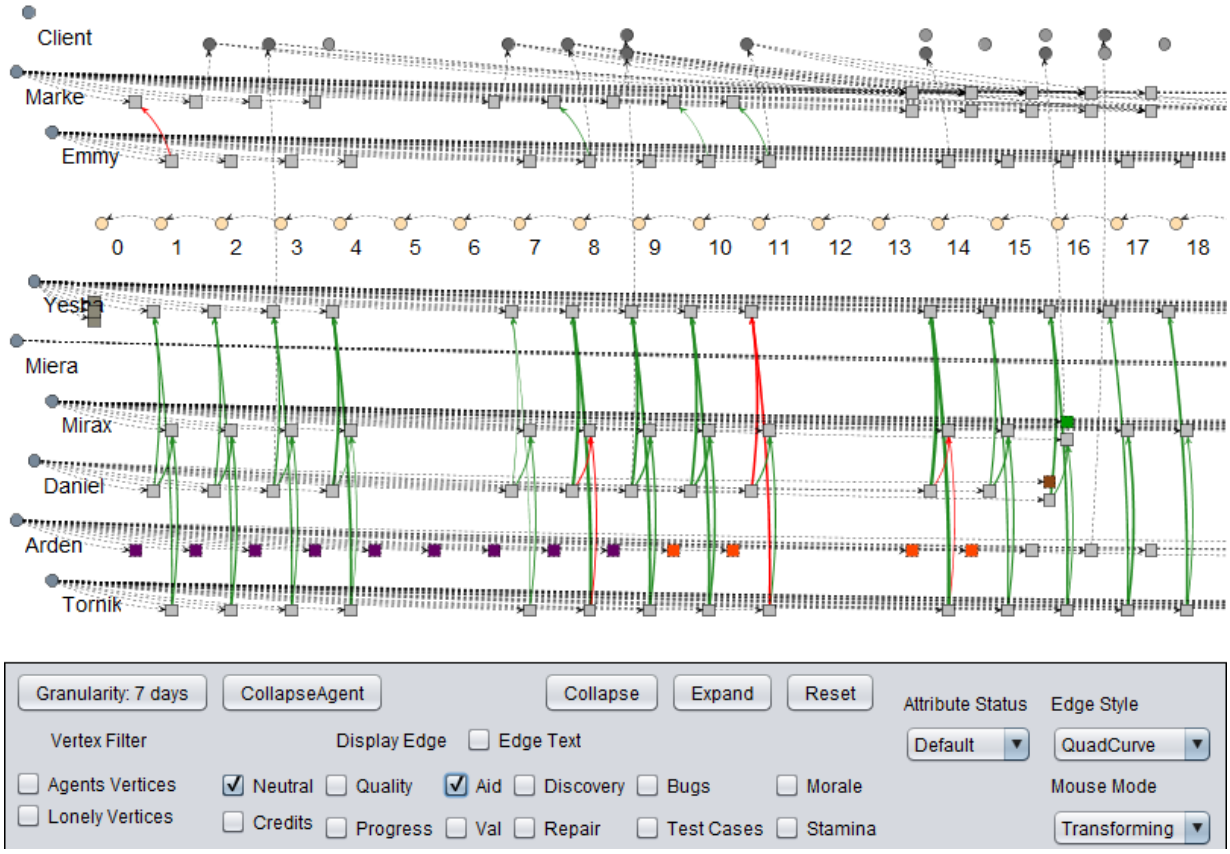


Figure 12: Prov Viewer's GUI instantiated for SDM.

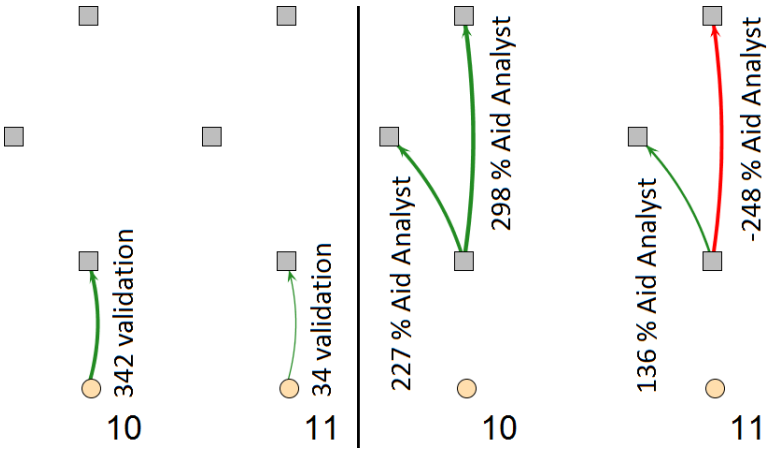


Figure 13: Analyzing the analyst's productivity.

Another usage of the display edge is to help to detect the reasons for drastic changes during the game. For example, detecting a major variation in the analyst's performance as shown in Figure 13, which dropped from *342* to *34 "requirement validation"*. The left picture has the "Val" edge display on, while the right picture has the "Aid" edge display on. The employees' roles in the figure are a manager (upper tasks), marketing (Middle), and analyst (bottom). The change in performance was detected by activating the "Val" edge filter and comparing the values (342 versus 34). The reason for this sudden drop can be traced to the

manager and marketing employees by changing the filter to “Aid”, which is a possible type of influence. By analyzing the displayed edges, the manager employee provided an aid of 298% in day 10 and a penalty of 248% at day 11 to the analyst due to wrong decision making. Moreover, the marketing employee provided a bonus of 227% and 136%, respectively for days 10 and 11. By combining these factors, at day 10 the analyst received a bonus of 525% in his task, while at day 11 he had, in total, a penalty of 112% for the execution of his task. The analyst productivity without any bonus was 65 at day 10 and 53 at day 11, which is within his productivity margin.

The “Attribute Status” changes the vertex color according to their values from the selected attribute. In SDM they can be Morale, Stamina, Hours (short for Working Hours), Weekend (highlighting “Saturday” and “Sunday” vertices), Credits, and Role. The vertex color does not change if it does not have the selected attribute. The default mode shows common activities with a shade of gray and uncommon activities with different colors. Common activities in SDM are normal tasks executed by employees during their roles, while uncommon activities are activities that do not happen frequently. The color difference amongst vertices is useful to quickly identity non-ordinary events. For example, by looking at the graph shown in Figure 12 it is possible to quickly identify that an employee was trained (purple vertices) during some days and was idle (red vertices) for a couple of days after the training was complete. This type of visualization, based on the evaluation of attributes, is useful to quickly identify particular sections in the graph.

With the available features, users can decide which type of edges and color schemes will be displayed on the provenance graph. Furthermore, the user can navigate in detail through the graph, exploring different sections of the game session or zoom out for a broader view. It is also possible to collapse sections of the graph to reduce its size and thus omitting vertices that might not be relevant for the current analysis. All graph manipulations can be reverted, and no information is lost during this process.

3.4 INITIAL FINDINGS

In our previous work (KOHWALTER; CLUA; MURTA, 2014) we wanted to investigate whether PinG was indeed a promising approach for game analysis, and to do so, we responded to these research questions:

1. Does provenance analysis help to understand events that emerged during the game?
2. Is provenance analysis more accurate than only watching a replay of the game session?

3. Is provenance analysis faster than only watching a replay of the game session?

We generated a replay of a game session and compared it with provenance analysis using a provenance graph to assess the possibility of using provenance analysis for improving understanding. This comparison was conducted through a questionnaire containing specific questions about events that occurred during the game session. Thirty-seven volunteers were divided into two groups: with and without provenance. Both groups watched the replay of the game session. The group with provenance also had access to the provenance graph. At the end, both groups answered the same questionnaire, which contained two questions related to the experiment execution (*i.e.,* starting time and group) and seven questions related to the events that occurred during the game, such as the motives that led an employee to quit during development or the main reason that allowed to complete the project in time. Our goal was to verify if the usage of provenance is beneficial and if it aids in the understanding of the events by giving access to provenance data from the game session.

We used two metrics to compare the results obtained by both groups: precision and time. The precision metric showed the correctness of the answers provided by both groups, which is related to understanding the factors that influenced the results. The time metric showed how long each volunteer took to answer all questions, thus allowing us to know if using or not provenance is faster. Furthermore, we performed statistical analysis over the results by means of hypothesis test to determine if the findings had any statistical meaning.

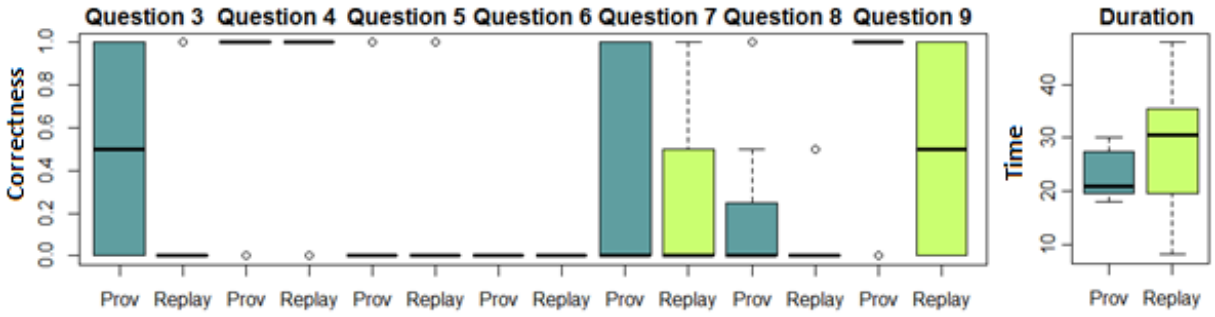


Figure 14: Brief summary of the results from (KOHWALTER; CLUA; MURTA, 2014).

Figure 14 presents a brief summary of the results of this experiment in the form of *box plots*. These results indicate, after running the statistical analysis, that in at least one case analyzing the game session with provenance is beneficial and provides equal or more correct answers than analyzing the game without access to provenance data, while also aiding in understanding the underlying influences between events and their effects. The other cases were not statistically different with the current sample size even when their mean values were greater

when analyzing the game session with provenance. However, that was sufficient to answer research questions one and two. Meanwhile, for research question three, the results clearly showed that analyzing the game flux with provenance was faster than analyzing without having access to provenance data, even when using the visualization tool for the first time and being unfamiliar with provenance concepts. These positive results opened a new door for exploring the usage of provenance concepts in the games domain, allowing richer and more abstract analysis of the tracked data.

3.5 FINAL CONSIDERATIONS

This chapter introduced perspectives on the game session comprehension process, showing that provenance data can induce deeper analysis and discussions regarding the game session. The previous findings showed that this knowledge can be used to help on (1) confirming the hypotheses formulated by developers, (2) extracting behavior patterns from individual sessions or groups of sessions, (3) understand the outcomes and (4) the causal relationships between events and actions.

The provenance graph also aids in the understanding of the events in the game by making explicit the cause-and-effect relationships between entities and actions. The provenance visualization allows the discovery of issues that contributed to specific game fluxes and results achieved throughout the game session. Thus, this type of analysis can be used to improve understanding of the game flux by identifying actions that influenced the outcome (WERNER et al., 2010), aiding the user to understand why they happened the way they did. However, the PinG approach provides only an abstract framework to act as guidelines for extracting provenance data and is necessary to adapt it for each game. Thus, it is necessary to manually create all the management and storage components, as well the tracking and gathering mechanism, and incorporate inside the game.

The initial findings from our previous work, presented in this chapter, gave us the necessary support and motivation to continue researching and advancing the usage of provenance in games to further improve understanding of the underlaying concepts and dynamics of a game session. All the concepts presented in this chapter were prototypes generated in our earlier work specifically for the SDM game and it would be necessary to generalize them so that they could be used in other games. These prototypes were used to verify the feasibility of this new field since it was the first time that provenance was being studied in this manner and being applied in games.

In the following chapter, we describe our proposal for providing a concrete framework to track causal relationships through provenance, which is based on the initial version of the conceptual framework presented in this chapter. This concrete framework is also game-independent, allowing it to be used in different genres. However, tracking provenance data is only the first step of the analysis process. As we discovered in our previous work, the visual representation of the data is also an important element to allow for designers to conduct exploratory analysis over tracked data and extract knowledge of what happened in the game session. Thus, we also present further advances made in our provenance visualization tool in the following chapter of this thesis.

CHAPTER 4 – GATHERING PROVENANCE FROM GAME SESSIONS

4.1 INTRODUCTION

Tracking game telemetry data and making it understandable is challenging due to the complexity of the games, leading to huge amounts of information. Moreover, deciding which information should be tracked and recorded is another challenge. One of the most common types of telemetry data is collected though states changes (LIU et al., 2011; WALLNER, 2013), (EL-NASR; NGUYEN, 2015). Even though state data is easier to examine, they lack contextual information and provides only a high-level view of what transpired in the game. In contrast, telemetry data that captures events (JOSLIN; BROWN; DRENNAN, 2007; KIM et al., 2008b) can provide more low-level and fine-grained information, capturing and describing player's activity and relating it more closely to the game session. Furthermore, since the data is collected at fine-grain, developers can use aggregating techniques to summarize the data by giving an overview of the game sessions and only digging through the fine-grained data when necessary.

However, no known approach used by the game industry for game analytics take into consideration the cause-and-effect relationships between events during a game session, which may be an important factor for determining the reasons that led to a certain outcome. In the previous chapter, we presented an earlier work (KOHWALTER; CLUA; MURTA, 2012) that introduced the usage of digital provenance in games in order to detect these cause-and-effect relationships. This novel approach resulted in another work (JACOB et al., 2014) that extracts provenance information using a non-intrusive technique through image processing mechanisms. Also in the previous chapter, we demonstrated an earlier evaluation (KOHWALTER; CLUA; MURTA, 2014) that brought some initial evidence of the benefits of using the PinG approach during game analysis of serious games, helping students to understand the underlying reasons for an outcome.

However, the PinG Conceptual Framework described in Chapter 3, requires a data infrastructure to store tracked data for the provenance visualization stage. It uses a list of actions for each character in the game. For example, each *agent* had a list that contained all actions executed by her. Furthermore, actions were connected to each action that influenced them. That data structure was simplistic, clean, and easy to understand. However, we noticed that it presented some problems with more dynamic games with multiple characters. By using that data structure, which stores the action list on the character itself, we needed to preserve the game object that contained the action list even after it ceased to exist in the game (*e.g.*, the

character died or left the scene). Furthermore, by the end of the session, there would be as many lists as characters that appeared in the scene. This becomes even more difficult when exporting the provenance data from the game to a dataset for analysis, requiring merging all lists in the same file or exporting multiple files, one for each character.

Moreover, the relationships between actions brought another problem: the conceptual frameworks do not define a storing and managing structure to deal with causal relationships. Thus, it is necessary an extra step to derive the relationships from the PROV provenance model (default provenance relationships) and the cause-and-effect relationships, which were stored on the affected action to inform which action affected it and how it was affected. Lastly, PinG provides a conceptual framework. Therefore, it is necessary to manually instantiate the approach and adapt it to the specific context of the game.

The main goal of this chapter is to propose major improvements over PinG and create a novel concrete framework, named PinGU, for capturing the provenance data and automatically generate the provenance graph for analysis. Unlike its predecessor, which was just an idea to capture provenance through a few guidelines, the now concrete framework PinGU offers support for capturing provenance data through automatic management and storage of provenance data. We implemented our provenance capture concrete framework in the Unity game engine, making the adoption of the PinG conceptual framework by existing games simpler. PinGU(KOHWALTER; MURTA; CLUA, 2017) is a low coupling solution that allows an easier integration of the idea behind PinG in existing games. We present our PinGU concrete framework in action by applying it to four different games, showing that we are able to gather provenance data and seemingly generate its graph.

The remaining of the chapter is organized as follows: Section 4.2 presents our proposed framework and improvements based on the conceptual framework. Section 4.3 shows four case studies over different games. Section 4.4 presents the related work. Lastly, Section 4.5 provides the final considerations for this chapter.

4.2 PING FOR UNITY

In the following subsections, we present our proposal for a new provenance tracking and managing framework, along with the new provenance gathering process. This concrete framework aims to provide a low-coupling solution for tracking provenance data, unlike the original conceptual framework from PinG. Furthermore, we describe our strategies for tracking cause-and-effect relationships using our concrete framework.

4.2.1 **PING FOR UNITY MODEL**

The PinG concrete framework for Unity (PinGU) is composed of classes written in UnityScript (a version of JavaScript used by Unity3D) that provides easier provenance extraction, requiring minimal coding in the game's existing components. PinGU has three different types of classes: eight *Core* classes, one *Interface* class, and five *Auxiliary* classes.

Figure 15 illustrates a simplified class diagram for our concrete framework, named PinGU (PinG for Unity). *Core* classes are in yellow, *Interface* classes are in light blue, and *Auxiliary* classes are in orange. The *Core* classes represent the infrastructure of PinG and are responsible for provenance information management, making everything transparent to the game designer. Analogously, it can be referenced as the provenance "server". Our concrete framework had been decoupled from the game, but for now it remains coupled with the engine itself (*i.e.*, Unity).

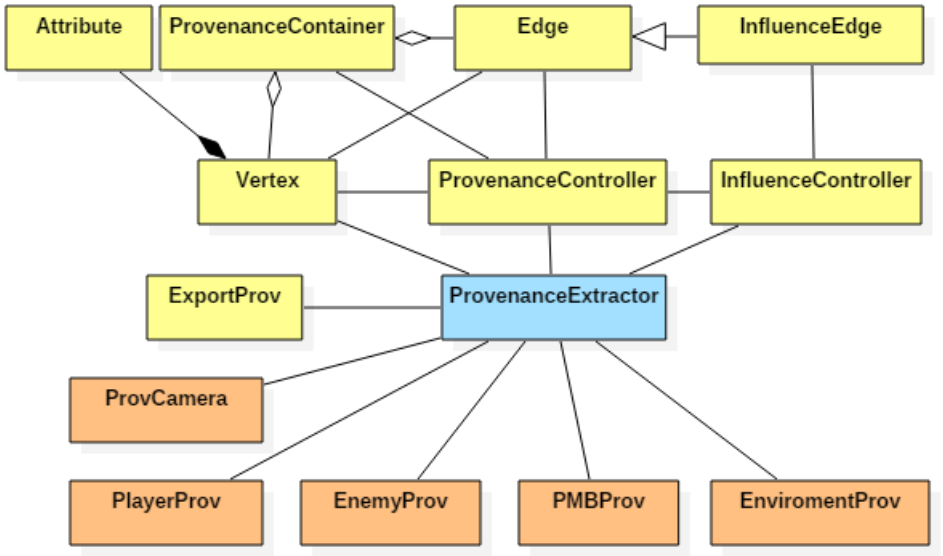


Figure 15: PinG for Unity class diagram

The concrete framework has only one class (*Provenance Controller*) responsible for storing all the provenance data, centralizing it in one list instead of having a list for each character. Furthermore, it uses two structures for storing provenance data: one list for storing actions, events, characters, and objects, which are the vertices of a provenance graph, and another list for relationships that appear as edges in a provenance graph. These two data structures (*Vertex* and *Edge*) store the provenance data in the format used by the graph and thus eliminates any extra steps when exporting the data. Meanwhile, the *Influence Controller* class manages the cause-and-effect relationships (influence edges), dealing with possible influences and passing it to the *Provenance Controller* class when they actually happen in the game. The

Provenance Container class is responsible for exporting the provenance data from the game to an XML file.

The *Interface* class (*Provenance Extractor)* is the gateway between the game and the *Core* classes. While the *Core* classes can be seen as the server, the *Interface* class can be seen as the client application and is responsible for tracking all provenance information and passing it to the *Provenance Controller* for storage.

Lastly, the *Auxiliary* classes contain domain-specific functions customized for a specific behavior or games and are responsible for gathering domain-data. These classes represent the provenance tracking functions that are used during provenance gathering and are required to be inserted in the existing game classes to capture telemetry data. Nevertheless, the existing templates can be used as a guiding example in cases that the desired action is not already implemented. These classes are *PBMProv*, *PlayerProv*, *EnemyProv*, and *EnviromentProv*, and each is customized for the particular genre of the game they represent (Car-related movements, Player, Enemy, and Environment). The *ProvCamera* class is related to capturing the game map, which is explained in more details in Section 4.2.5.

4.2.2 **PROVENANCE GATHERING**

The PinG for Unity approach is responsible for gathering all provenance data through its *Provenance Extractor* class and passing them to the *Provenance Controller* when the data is ready to be stored. Thus, instead of each character having a list of actions, now the character only needs to notify the *Provenance Extractor* class the action executed together with any additional desired information (*i.e.*, attributes), as illustrated by Figure 16. After receiving the notification, the *Provenance Extractor* class packs the details of the execution, creates the corresponding vertex (Activity, Agent, or Entity), and sends it to *Provenance Controller*, which in turn stores the new vertex in the list for vertices. Default provenance relationships are automatically generated by the Provenance Controller, which are the relationships between activities and the agents that executed them, along with the correct chronological order of events between actions. In other words, these are all relationships that are not related to influence (*i.e.*, cause-and-effect). Capturing influences in a game require a more complex procedure, which is explained in the following subsection.

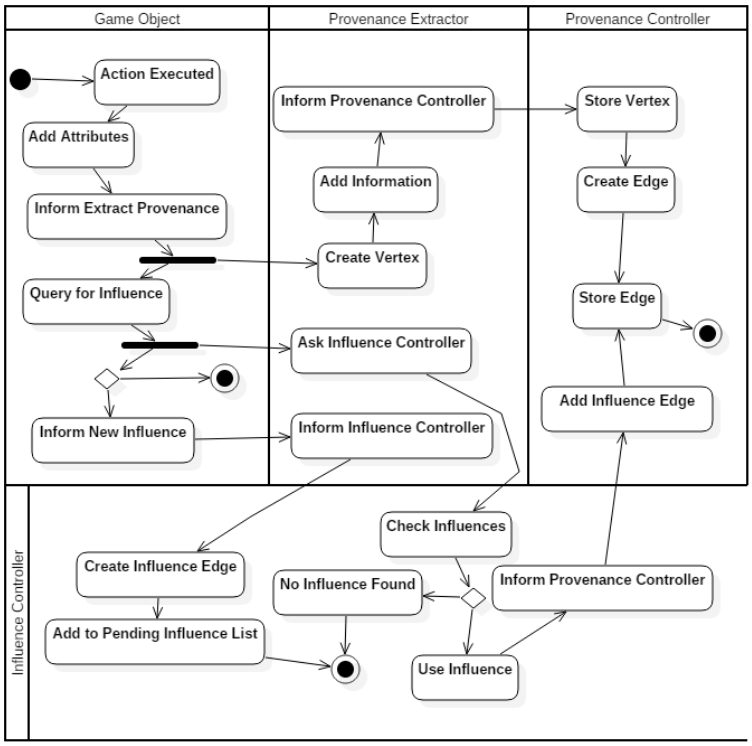


Figure 16: PinG for Unity tracking provenance data

4.2.3 **TRACKING INFLUENCES**

The procedure to track cause-and-effect relationships, or influences, is similar to the one for storing the execution of an action, which notifies the *Provenance Extractor* class. When an action is executed and notified to *Provenance Extractor*, the game should also notify if the action generated any possible influences. If it does, then an influence is generated and the *Provenance Extractor* passes the information to *Influence Controller*, which is the class responsible for managing all influences in the game. After receiving the notification, *Influence Controller* creates a special *Influence Edge* and stores it in a pending influence list until it is used. Furthermore, *Influence Controller* registers the point of origin (an action that generated the influence) and the trigger (when this influenced is used) for the influence. If the influence has any expiration time (*e.g.*, turns) or a limited number of usages (*e.g.*, for the next five attacks), then the *Influence Controller* also manages these limitations, removing the influence from the pending influence list when it expires. Therefore, it is necessary to notify possible influences that the action generated after notifying the action execution in order to track influences. Figure 16 also illustrates the necessary steps taken in order to track influences in the provenance graph. Each lane represents a different class from our framework, with the exception of the "Game Object" that is the object in the game (*e.g.*, player) that executed the action.

Unfortunately, information related to influences needs to be present in the game design, explicitly saying if an action can affect other actions, such as an effect of a spell that will affect the player that used it. A trivial example is when a character causes and takes damage during the game. For instance, an attack action can generate a damage value if the attack hits the target. Thus, when the attack action is executed and notified to *Provenance Extractor*, it is also required to notify the possible damage influence. The damage influence only takes effect when the target is hit. At that moment, it is necessary to query for possible influences, which in this case is damage. This query is done through *Provenance Extractor* by asking if there are any pending influences that satisfy the criteria or trigger (in this case, take damage), which in turn asks the *Influence Controller* to verify it. If the *Influence Controller* finds a pending influence that satisfies the criteria, then it updates the influence (decrease counter or remove it from the list) and passes the information to *Provenance Controller* to create the corresponding edge in the graph. Then *Provenance Controller* links the influence on the current action and the action that generated the influence, which is known by *Influence Controller,* along with any other details about the influence. If there are no pending influences that satisfy the criteria, then nothing is passed to *Provenance Controller* and no influence edge is created.

4.2.4 **TRACKING LOST INFLUENCES**

In order to record influences that were missed during the game, such as when the player did not take an item or allowed an enemy to escape, it is required also to inform the desired target of this influence (*e.g.*, player). Otherwise, the *Influence Controller* would not know the target of the "missed" influence. For example, consider a situation where there is an item in the scene and we want to know if the player took it or not. This can be accomplished simply by verifying if the player took the item or not during the game. However, we want to clearly show to the designer that the player did not take it, without the need to infer it from the lack of data. This is important because it could affect an analysis of the player performance in the level, or discovering the reason behind her death (*e.g.*, she missed a health item). Thus, when the item is placed or spawned in the scene, it notifies its existence to *Provenance Extractor* (to create the entity vertex for the item) and the possible influence of restoring health when taken by the player. Moreover, it also needs to inform to the *Influence Controller* the need to consider the influence when the player misses the item. In this case, the desired target for this influence is the player. With this extra information, when the player skips the item, *Influence Controller* can create an influence stating that the player did not interact with the item. If the player took the item, then it creates the regular influence (which in this case is healing the player).

4.2.5 **CAPTURING GAME MAP**

We also implemented a specialized camera class in order to simplify the process of capturing the game map to use it in combination with the provenance graph. This camera is orthographic, which preserves the dimensions and does not change coordinates to accommodate the perspective of the viewer. Thus, the camera needs to be positioned either directly above the game scene or laterally (for platform games), allowing it to capture the entire map. This class automatically captures the screenshot of the scene and the necessary data required to align the provenance graph, which uses world space coordinates, with the captured map, which uses pixel position, not requiring manual compositions. The screenshot resolution can also be adjusted in the class. This class allows to easily superpose the provenance graph over the map of the game to improve the analysis process by linking events to game locations.

The camera class captures the camera's world position (*cameraPosition*) and the camera's upper left corner coordinates in the world position (*leftCorner*). The camera's position is used to translate the game map in order to align it with the graph and is easily obtained by getting the position of the camera in the world space. The second information is used to scale the graph to match the picture and is captured by converting the camera position from viewport space to world space, which is the upper left corner.

In order to align the graph with the map, it is necessary to find a scale factor, that can be trivially calculated by Equation 1. The equation uses half the screenshot's picture width to determine the distance between the center, which is the position in the picture where the camera is when the screenshot was taken, to the left edge to properly scale the graph.

Equation 1: Scale Factor to align the graph with the game map

$$scaleFactor = \frac{0.5 \times pictureWidth}{leftCorner.x - cameraPosition.x}$$

The *scaleFactor* is used to transform the world coordinates used in the provenance data to pixel coordinates used in the screenshot of the game map. Therefore, the game designer only needs to position the orthographic camera in the game scene and add the camera class in order to capture the entire map and the necessary data. After that, the designer can use the coordinates captured by the class and the screenshot in a visualization tool.

4.3 INTEGRATING PINGU INTO EXISTING GAMES

A game developer can use PinGU, which is available at GitHub[7], to capture provenance data from a game by following four stages described in this section: (1) adding the provenance controllers in the scene, (2) analyzing the game design document to understand what knowledge for the provenance tracking will be extracted, (3) attaching the provenance extractors in each agent, and (4) creating and attaching the provenance tracking functions.

The following sub-sections present three open-source game samples (*2D Platformer Tutorial*[8], *Car Tutorial*, and *Angry Bots*[9]) that we integrated PinGU into the game to track provenance data, for validation purpose. We also show an in-house game where the adoption of PinGU was done by a third party. The games were not modified in any way nor added new features besides coupling with PinGU, which is only responsible for tracking provenance data. In this section, we only show the PinGU integration steps for each game. The next chapter, which is responsible for provenance visualization, provides some analysis examples of these games using the provenance data generated with PinGU.

4.3.1 2D PLATFORMER TUTORIAL

We use the game *2D Platformer Tutorial* from Unity as our first running example of the PinGU integration. Figure 17 shows a screenshot of the game where the player has to kill aliens to gain score points. The game has two different types of enemies and the player can collect two different types of items to aid in his fight (health and ammunition items).

The **first stage** of usage consists of creating a game object in the scene to act as a centralizing server for the provenance information. This game object will have two attached classes, *ProvenanceController* and *InfluenceController*, which are illustrated in Figure 18. As said earlier, both classes are used to manage all provenance information and graph generation, thus, only one instance of each are necessary per game scene. If the game is comprised of multiple scenes, then each scene will have its own provenance graph. These two classes use the other *Core* classes, which act as libraries and must not to be placed in the scene.

---

[7] http://gems-uff.github.io/ping/

[8] https://www.assetstore.unity3d.com/en/#!/content/11228

[9] https://www.assetstore.unity3d.com/en/#!/content/12175

Figure 17: 2D Platformer game.

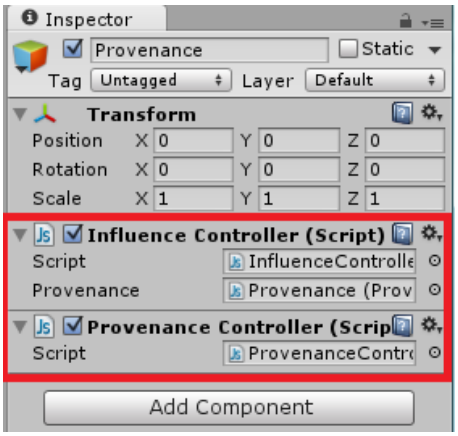


Figure 18: 1st stage for PinGU instantiation, showing the *Provenance* game object and its scripts.

The **second stage** is to identify the actions and their interactions with other actions in the game design document. In the running example, we identify the existing classes that contain the actions that we want to track, which are illustrated in Figure 19. The same figure also shows a summary of each selected class and their responsibility in the game, grouped by the identified agents (*i.e.*, *Enemy*, *PlayerControl*, *PickupSpawner*). The classes for the agents also contain additional actions, such as spawning item and movement.

The **third stage** is to attach the *ProvenanceExtractor* class in each character or entity in the game (*i.e.*, NPCs, player, interactive objects, prefabs) and link it to the object created in the first stage. This class is responsible for creating all the provenance vertices for the game entity that is attached to and then passing these vertices to the *ProvenanceController* to insert it in the graph. Figure 20 illustrates an example of adding the class to the *Hero* game object, which is the player's avatar from the 2D Platformer game.

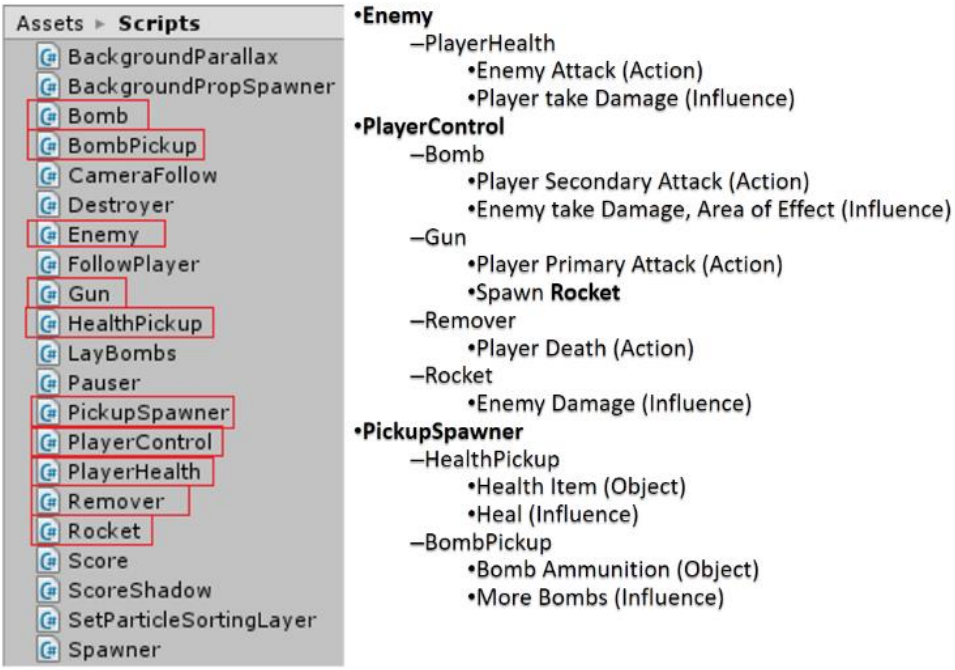


Figure 19: the 2nd stage for PinGU integration, showing the 2D Platformer classes and Game Design.

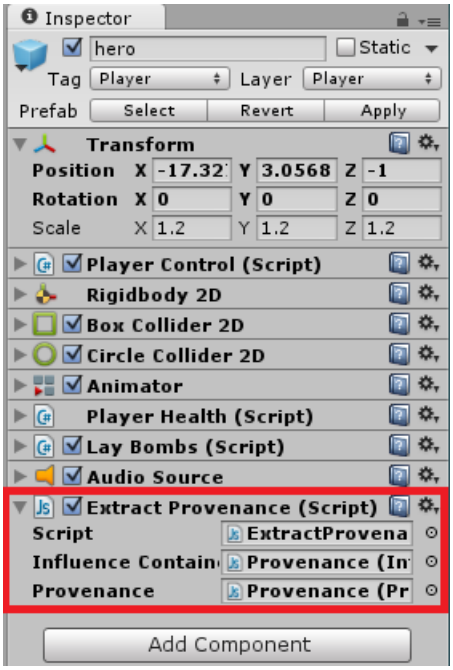


Figure 20: the 3rd stage for PinGU integration, showing the insertion of the provenance tracking class in existing agents and entities.

The **fourth stage** is creating the domain-specific *provenance tracking functions* and attaching them to each entity in the game that has the *ProvenanceExtractor* class. Each existing class should have a *provenance function* for each possible action that the entity can perform and that we are interested in tracking. It is also important to create the agent vertex for each entity that has the *ProvenanceExtractor* class, which is easily accomplished by adding the "newAgentVertex" function inside the "MonoBehaviour.Awake" area.

Unfortunately, it is necessary to create these *provenance functions* due to domain contextual information. However, all these *provenance functions* are small and simple, following the same four-step recipe and changing only the context information used during each step:

1. Add game-related attributes (*e.g.*, health points, experience points, etc.);
2. Create the appropriate vertex (Activity, Agent, or Entity);
3. Check for influences (if applicable);
4. Generate influence (if applicable).

Step A is used to configure the desired information to be extracted during the execution of each action or event. They will appear at the graph's vertices as attributes. Unity already provides default attributes, such as location, tag, and object name. However, game-sensitive attributes such as health points, magic points, and player score must be manually added by the *AddAttribute*(<name>, <value>) function of *ProvenanceExtractor* class. After adding the desired attributes, the **step B** creates the provenance vertex and places it in the graph. This vertex can be any of the three provenance types and must be specified by the user by calling the *NewActivityVertex*, *NewAgentVertex*, or *NewEntityVertex* functions.

Steps C and D are related to influence. The third step is used to verify if there is any influence that can affect the current action. If so, they are automatically inserted in the graph as an edge connecting the respective vertices. This verification can be made by a tag (*HasInfluence(<tag>)*), which is used to group a collection of influences that has something in common, or by an influence ID (*HasInfluence_ID*(<ID>)). Both steps can have multiple instances of each. For example, one action can generate three different influences and be affected by two.

Step D is responsible for creating influences (*GenerateInfluence*), so they can be used by **step C**. Influences can be created with some restrictions: They can expire when a certain time passes (*e.g.*, spell duration), leading to the E (expire) suffix at the function (*i.e.*, *GenerateInfluenceE*), or after a number of times used (*e.g.*, spell that block the next X attacks) leading to the C (consumable) suffix (*i.e.*, *GenerateInfluenceC*), or both (*GenerateInfluenceCE*). Lastly, the missed influences can be combined with the restrictions above, which represents something that was expected to happen but for some reason, it did not. For those, the function has the suffix M (“missed”) (*i.e.*, *GenerateInfluenceMC*, *GenerateInfluenceMCE*).

Code 1 shows an example of a *provenance function* for our running example of one of the possible actions that can be executed by an enemy. The calls used in the *Prov_Attack* function are implemented in the *ProvenanceExtractor* (*NewActivityVertex*, *HasInfluence*, *GenerateInfluenceCE*) class, with the exception of *Prov_GetEnemyAttributes*, which is domain related and the developer need to specify the desired attributes for tracking, besides the default attributes from Unity (*i.e.*, Tag, object name, object coordinates). This is accomplished by creating a function (*e.g.*, *Prov_GetEnemyAttributes* from the auxiliary classes) that invokes the function *AddAttribute* from *ProvenanceExtractor* by passing the attribute name and value for each attribute, as also illustrated by Code 1.

```
public string Prov_Attack(float damageAmount)
{
    Prov_GetEnemyAttributes();
    prov.NewActivityVertex("Attacking","");
    prov.GenerateInfluence("Player", this.GetInstanceID().ToString(),
    "Damage", (-damageAmount).ToString());
    return this.GetInstanceID().ToString();
}
// Enemy Attributes
public function Prov_GetEnemyAttributes()
{
    prov.AddAttribute("Health", hp.health.ToString());
}
```

Code 1: PinGU code for tracking game data. Orange text in the code is domain-related.

After creating the necessary *provenance functions* for their respective game objects, the next step is to incorporate the function calls in existing game classes in order to register the provenance information. All this process becomes trivial if the developers have a detailed game design document stating all the possible actions that can be executed in the game along with their purpose. The action list shows the actions that are desired to be tracked and the necessary *provenance functions* that need to be made. Meanwhile, the action's purpose gives us insights on the influences that they can generate during or after executing the action.

Code 2 shows an example of code insertion in an existing game class responsible for controlling the artificial intelligence (AI) of enemy characters in the game. The "damageAmout" is a configurable variable from the original class that states the damage the attack will cause. We inserted a call to the *Prov_Attack* function, whose code appears in Code 1, in the function responsible to make the enemy AI fire at the player.

```
function Fire () {
    if (weaponBehaviours[nextWeaponToFire]) {
        weaponBehaviours[nextWeaponToFire].SendMessage ("Fire");
        nextWeaponToFire = (nextWeaponToFire + 1) % weaponBehavi
        lastFireTime = Time.time;

        // Provenance
        prov.Prov_Attack(damageAmount);
    }
}
```

Code 2: *Provenance function* call insertion into existing classes.

```
public void Prov_Death()
{
    Score score = GameObject.Find("Score").GetComponent<Score>();

    prov.AddAttribute("Health", "0");
    prov.AddAttribute("Score", score.score.ToString());
    prov.NewActivityVertex("Death","Drowned");
    Prov_Export();
}
void Prov_Export()
{
    Debug.Log ("Exported");
    GameObject ProvObj = GameObject.Find("Provenance");
    ProvenanceController prov = ProvObj.GetComponent<ProvenanceController>();
    prov.Save("2D_Provenance");
}
```

Code 3: *Provenance function* for the player's death action.

```
void OnTriggerEnter2D(Collider2D col)
{
    // If the player hits the trigger...
    if(col.gameObject.tag == "Player")
    {
        // .. stop the camera tracking the player
        GameObject.FindGameObjectWithTag("MainCamera").GetComponent<CameraF

        // .. stop the Health Bar following the player
        if(GameObject.FindGameObjectWithTag("HealthBar").activeSelf)
        {
            GameObject.FindGameObjectWithTag("HealthBar").SetActive(false);
        }

        // ... instantiate the splash where the player falls in.
        Instantiate(splash, col.transform.position, transform.rotation);
        // ... destroy the player.
        Destroy (col.gameObject);

        //Provenance Death
        Prov_Death(col.gameObject);

        // ... reload the level.
        StartCoroutine("ReloadGame");
    }
    else
    {
        // ... instantiate the splash where the enemy falls in.
        Instantiate(splash, col.transform.position, transform.rotation);

        // Destroy the enemy.
        Destroy (col.gameObject);
    }
}
```

Code 4: Fragment of the original *Remover* class: Added the *provenance function* in the player's death.

The last step is to add a provenance export function to an event, so it can save the current provenance graph to an external XML file when the designated event is executed (*e.g.*, player's death, completing the level, etc.). Code 3 illustrates the *provenance functions* for our running example responsible for exporting the tracked data, which is linked to the player's death, and Code 4 shows the insertion of the *provenance functions* to track the information.

The PinGU integration is explained in more detail in the tutorial available at the concrete framework GitHub page, showing all *provenance functions* and their insertion in the identified classes. Figure 21 shows an example of the generated provenance graph from the tracked actions executed during a game session. We can see in this graph the player's and each enemies' actions and how they interacted with each other by looking at the vertical colored edges. However, this type of visualization will be discussed in detail in the next chapter.

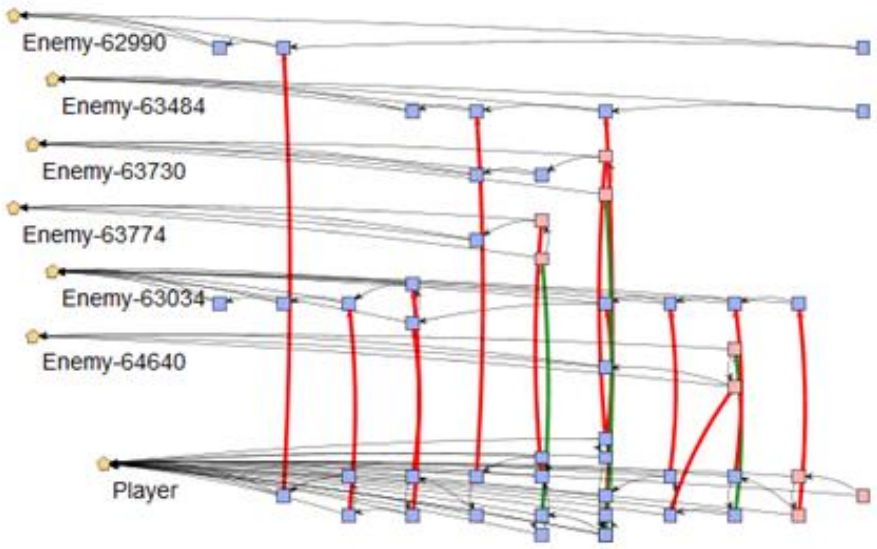


Figure 21: Example of the generated graph for the *2D Platformer*.

4.3.2 **CAR TUTORIAL**

The second case study is the Car Tutorial from Unity asset store. This tutorial has only one racetrack and focuses on the arcade style racing game. In addition, there is no implemented AI for opponent cars. Incorporating the framework in the game results in PinGU tracking events and actions executed by the player's car during the game session, along with their effects on other events, to compose the provenance graph (*e.g.*, crashing the car, pressing the car's brake, etc.). This is an interesting and complementary case study because the game is from a completely different genre from the previous one.

Figure 22: Screenshot of the Car Tutorial from Unity

Incorporating PinGU in this game is trivial because it only has a single agent, which is the player. Following the stablished steps in the previous section, we start by importing PinGU into the project's Plugins folder in the **first stage**. Then, we create an empty game object in the scene with the name Provenance so we can add both provenance controllers in it, as shown in Figure 23.

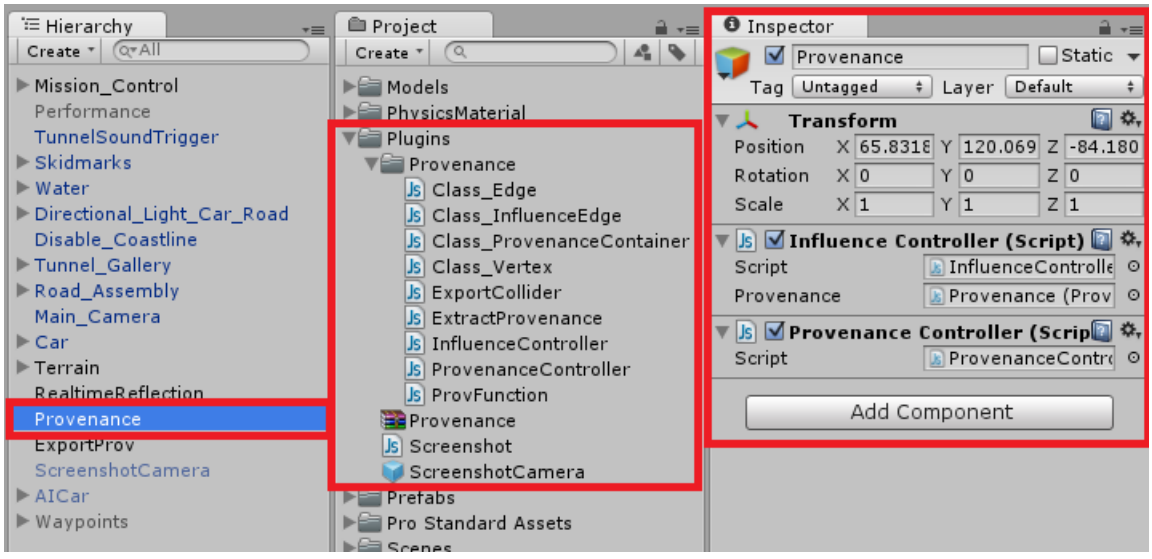


Figure 23: Car Tutorial's stage 1 for incorporating PinGU

Stage two is responsible for analyzing the game design document to find out the elements involved in the game, possible actions and events, as well as the impact each one can have during the game. Unfortunately, there is no game design document for this game, since it is a tutorial to demonstrate how to assemble a very basic racing game. Nevertheless, we

identified the agent and all possible actions by looking at the code, guided by the tutorial manual.

Stage three is very short in this game since there is only the player's car. Thus, we added the *Provenance Extractor* script in the "Car" game object. **Stage four** is the most complex and work-demanding stage for incorporating PinGU. In this stage, we created all the provenance tracking functions related to this game. For simplicity, we concentrated all the functions into a single script and attached it to the "Car" game object. First, we create the agent vertex for the "Car". This is easily accomplished by adding the "*newAgentVertex*" method inside the "*MonoBehavour.Awake*" function of the "Car". Thus, the player's agent vertex will be automatically created whenever his car spawns in the game. The agent vertex is important because it links all the player's actions in the provenance graph.

Still, during the **fourth stage**, the next step is to add the function calls inside each function related to action execution or event that can happen, which are concentrated into only two scripts in this project: "Car" and "Crash Control". The later only have one function which identifies when a car crash into an object. We identified eight possible actions in this game, which we added their respective provenance tracking calls: (1) Hand Brake, (2), "Release Hand Break", (3) "Flip", (4) "Landing", (5) "Flying", (6) "Change Gear", (7) "Braking", and (8) "Crash". Some of these actions had their own functions, while others were implicit in the code while updating the game state (*i.e.*, landing, flying, braking). Each function basically follows the four previous stablished steps. Some of them required extra minor coding to make additional logic checks to avoid modifying the game's existing scripts besides insertion calls. These functions are available at the "PMBprov" class.

Figure 24 illustrates the Car's game object after **stages three and four**, showing the provenance tracker (*Provenance Extractor*) and provenance domain functions script (PMBProv) and Code 5 shows an example of existing function in the game (Flip) and its respective provenance domain function (Prov_Flip) that tracks that information. Note that Prov_Flip can be affected by two different types of influences: "Player" and "Flip". The first will check if there are any previous actions that generated any general influence that affects this agent, independently of the currently executed action, and the second checks if there is any influence related to this particular action. One example that generates the second type of influence is when the car crashes in an obstacle, which sometimes can result in a flip and activate the Flip function. Thus, the flip happened only because the agent crashed the car and that causal relationship is captured.

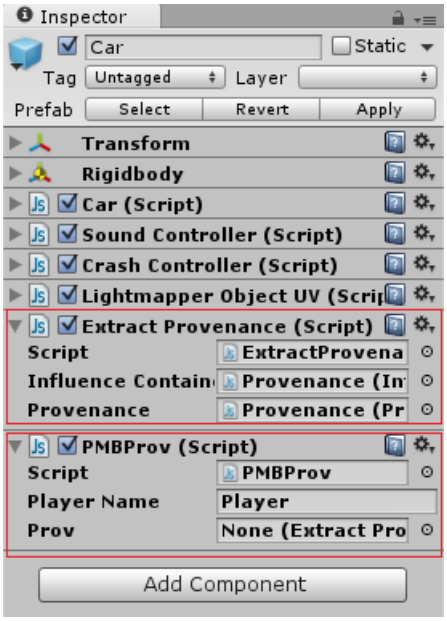


Figure 24: Car Tutorial's stage 3 and 4 for incorporating PinGU

```
function FlipCar()
{
   transform.rotation = Quaternion.LookRotation(transform.forward);
   transform.position += Vector3.up * 0.5;
   rigidbody.velocity = Vector3.zero;
   rigidbody.angularVelocity = Vector3.zero;
   resetTimer = 0;
   currentEnginePower = 0;

   // Provenance
   prov.Prov_Flip();
}
public function Prov_Flip()
{
   Prov_GetAgentAttributes();
   prov.NewActivityVertex("Flipped");
   prov.HasInfluence(agentName);
   prov.HasInfluence("Flip");
}
```

Code 5: Car Tutorial's provenance call example

Lastly, we added an empty game object to export the provenance data when the player completes a lap. We also added the screenshot camera to capture the entire track to link the provenance data to the game regions.

4.3.3 **ANGRY BOTS**

We conducted a third case study using a very different style of game, called Angry Bots, also from the Unity asset store. Angry Bots belongs to the hack-and-slash genre, being a top-down action shooter. In the available scenario, the player must face enemy robots and interact with the environment in order to complete the level. Figure 25 illustrates a typical screen of the game.

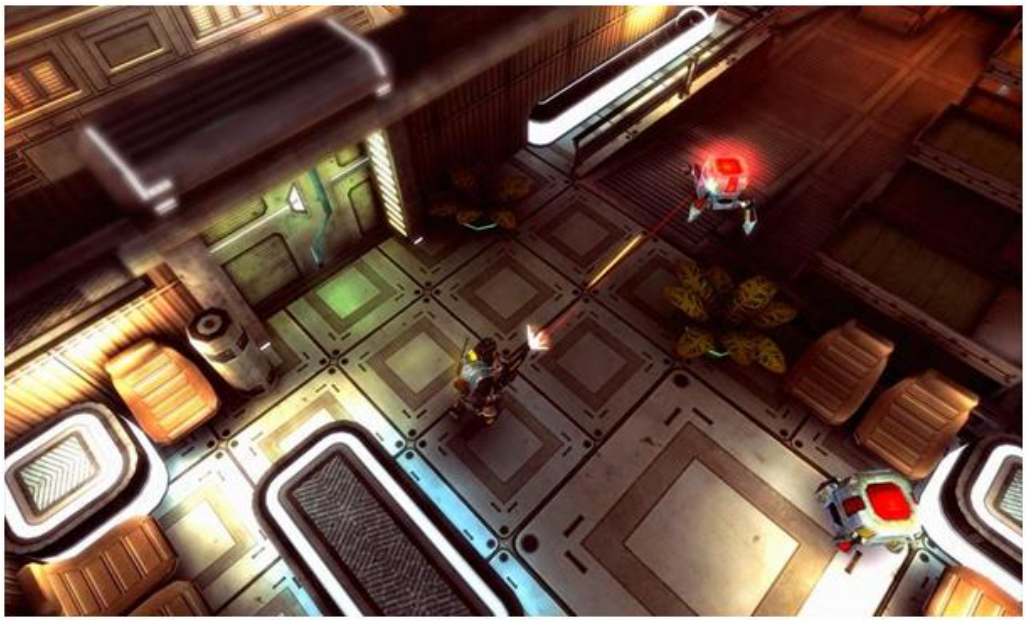


Figure 25: Screenshot of the Angry Bots game from Unity

This game is much more complex than the previous Car Tutorial, providing a full game level with multiple enemies and objects to interact with. Nevertheless, the PinGU incorporation follows the same four stages after importing it into the project. In the **first stage,** we create the Provenance game object and attach the provenance managing scripts, in exactly the same way we previously did.

In the **second stage,** we analyze the game document to determine all the agents and actions available in this game. Once again, as this was a tutorial game, there was no formal game design documentation. Nevertheless, we looked at the tutorial manual for this game and the available scripts to piece together all the necessary information for successfully tracking provenance data.

The **third stage**, despite being a more complex game than the previous one, is also trivial. During the second stage, we identified all agents: four different types of enemies (*Spider*, *Buzzer*, *Mech*, *ConfusedMech*), the *player*, and the *environment*, which controls computers that unlock doors. We then added the provenance tracker (*ProvenanceExtractor* class) in each agent's prefab[10].

The **fourth stage** requires to insert tracking calls in each action of the game, as well as create their corresponding provenance tracking function following the same recipe. First, we

[10] A prefab asset in unity acts like a template to create new instances of an object in the scene.

need to create the agent vertex for each agent that we identified at the third stage. Once again, this is accomplished by adding the corresponding function inside the "*MonoBehaviour.Awake*" function.

Continuing in the fourth stage, we identified nine different actions in the game: (1) Hacking, (2) Death, (3) Respawn, (4) Attack, (5) Take Damage, (6) Explode, (7) Regenerate, (8) Spot Player, and (9) Lost Track (of player). We then proceeded to create their respective *provenance functions* following the same previously presented four-steps recipe. All these domain-specific *provenance functions* are available at the three domain-specific classes that appear as examples in Figure 15: *PlayerProv*, *EnemyProv*, and *EnviromentProv*. Lastly, we added these *provenance functions* in their respective functions that execute the respective action.

Unlike the previous case, this game has interactive game objects in the form of computer terminal and doors. These interactive game objects appear in the provenance graph in the form of *Entity* vertices. Thus, each interactive object needs to have an *Entity* vertex, which can be easily done similarly to creating vertices for agents by adding the "*newEntityVertex*" function whenever the object spawns. Thus, the provenance tracker will be able to record the action of interacting with the object and link it to the interacted object when the player interacts with an object. For example, when the player interacts with a computer terminal, then the tracker created the "Hacking" action, linking this action to the computer terminal through a causal relationship edge. The hacking action makes the computer terminal to unlock a door in the scene after the hack. This unlocking is also recorded in the form of a causal relationship between the computer terminal and the door it unlocked when it was hacked by the player's action of "Hacking". Consequently, following the transitive closure of these links, we can infer that the player was responsible for unlocking the door.

4.3.4 **MORHWING**

The *MorphWing* game was developed in-house by students using the Unity3D game engine and integrated with our PinGU concrete framework. It's a simple 2D game where the main objective is to get the highest score by destroying enemies. Items are spawned throughout the game session, which can positively or negatively affect the player. This case study differs from the others since its conceptual development took into consideration the inclusion of provenance in its earlier stages. A screenshot of the game is presented in Figure 26.

Figure 26: Screenshot of the *MorphWing* game

The PinGU incorporation follows the same four stages after importing it into the project. In the **first stage,** they created the Provenance game object and attach the provenance managing scripts, in exactly the same way we previously did.

In the **second stage,** they analyzed the game document to determine all the agents, actions, and possible causal relationships in the game. From the document, they could see that in *MorphWing* the player loses health points while colliding with the enemy or being hit by enemy projectiles. In both cases, the player becomes invincible for a very short moment, nullifying all damage in this period, in order to give a chance for the player to recover. *MorphWing* presents four different enemy types with a maximum of four enemies at the same time. This value was chosen as it presented a good balancing challenge in the initial development tests. The characteristics and behavior of each type of enemy are as follow:

1. *Straight*: moves in a straight direction until it reaches the other side of the screen, disappearing afterwards.
2. *Chaser*: chases the player for colliding, causing damage, until gets destroyed.
3. *Boomerang*: after appearing in a random corner of the screen, moves straight for a moment, stops, shoots a bullet on the player's direction, and moves back towards its spawn point, disappearing afterwards.

4. *Round* Shooter: after appearing in a random corner of the screen, moves straight for a moment, stops, then shoots bullets clockwise in eight directions on a circular pattern, starting by the top.

Items are spawned and remain visible for four seconds and are removed if not collected by the player. In total, up to three items can be on the screen simultaneously. Each item has a different icon as well as a color, with green or red tinted for positive and negative effects, respectively. The following items are available in the game:

- *Healing*: recover a portion of the player's health.
- *Control Reverser*: temporarily inverts the player's movement directions (up becomes down and vice-versa) and left becomes right and vice-versa).
- *Damage* Up: temporarily increases the player's damage output.
- *Speed Up*: temporarily increases the player's movement speed.
- *Speed Down*: temporarily decreases the player's movement speed.

The **third stage** involves adding the provenance tracker class (*ProvenanceExtractor*) in each agent's prefab, which includes each enemy type, the player, and the agent responsible for spawning items. The **fourth stage** involves adding the functions to create the agent vertices for each identified agent in the third stage. Then, we insert tracking calls in each action of the game, as well as create their corresponding provenance tracking function as previously explained. We identified six different actions in the game aside from spawning each agent: (1) Spawn Item, (2) Death, (3) Attack, (4) Take Damage, (5) Heal, and (6) Power Up. The "Spawn Item" is the action that the Item Spawner's agent does when it creates a new item in the game. This is an activity-type action that generates another entity-type object. We then proceeded to create their respective *provenance functions* following the same four-steps recipe previously presented and inserting their calls in their respective sections.

4.4 RELATED WORK

The literature adopts different terms for **tracked game data**, such as gameplay data, logged data, play traces, and telemetry data. Moreover, the process of analyzing such data, referenced here as **game analytics**, is also named in different ways, such as gameplay visualization, visual data mining, and game session analysis. In this section, we kept the original terms of each work, as they are usually reflected in the approaches' names.

Joslin (2007) proposed the *Gameplay Visualization Manifesto* (GVM), which is a framework for gameplay data logging that uncovers gameplay events by attaching logging methods in game objects responsible for generating relevant events during the game. The event model is the basis for the game data logging framework. It encapsulates the information that is desired by users and classifies the events into three groups: immersion, quest, and social. The immersion group represents events related to increasing the player's sensation of being involved in the game flux. The quest group represents events related to quest creation, execution, and analysis. Lastly, the social group represents events related to social factors in the game, such as group meeting or interaction with other characters.

The main application of GVM is for collecting game metrics, such as player deaths, position, time spent in available features (e.g., crafting and fighting), item usage (e.g., equipment), actions performed, and player enjoyment. Therefore, GVM does not track cause-and-effect relationships, only the executed actions along with their timestamp and location, in addition to character attributes and equipment.

Kim *et al.* (KIM et al., 2008b) proposed the *Tracking Real-Time User Experience* (TRUE) approach that combines human-computer interaction (HCI) instrumentation, which collects *user initiated events* (UIEs) and logs file analysis techniques in order to automatically record user interactions with games. Thus, TRUE can capture behavioral data and the attitudinal information behind the decisions made by the player in order to obtain a better understanding of the context of each captured behavior.

Nevertheless, the designer still needs to infer the reasons behind the elements that led to an outcome. This occurs because the contextual information is only extra attributes that were tracked during the execution of the action and not actual relationships between events, and thus it does not capture the cause-and-effect relationships, that must be inferred by the designer when analyzing the logged data. Moreover, TRUE was designed for the industry and is not easily available for indie companies. Even though we did not explore attitudinal data with PinG, it can be trivially incorporated in our approach as attributes for the player's actions or by creating specific activity vertices only for the attitudinal data when they are captured.

Playtracer (LIU et al., 2011), which is a visual tool designed to illustrate how groups of players move through the game space, aids the designer by tracking game states and showing common pathways and alternatives that players used to succeed or fail in their tasks, identifying pitfalls and anomalies in the scene. Nonetheless, *Playtracer* does not consider temporal information and does not preserve the order of the states visited by players when he/she revisits

the same state. Moreover, incorporating *Playtracer* in the game design is challenging because it requires designers to define a state distance metric and identify relevant states.

Play-Graph (WALLNER, 2013) captures and illustrates the sequence of states and the actions that caused the state change in the players over the course of the game. In the Play-Graph context, a game state describes a certain configuration of the game or an entity, while actions consist on the player interactions with the game, such as shooting, jumping, or using an object. In this concept, a game is viewed as a finite state machine with a finite number of states and transitions between them. The states are composed of a set of attributes from the game and players trigger actions at some specific points in the game. However, due to the nature of how the data is structured in Play-Graph, the understanding of player behavior is guided by the player progression in the game (*e.g.*, killed a boss), and not by how he/she interacted with the world (*e.g.*, combat rounds from the battle against the boss). From the available documentation, there is no way to determine interactions or influences. Only the changes from one state to another, caused by an action executed by the player, can be identified. Conversely, influences in the player's action, such as an influence from another character that affected the transition of one state to another, are not present in the graph (there are no edges linking edges).

More recently, Loh and Sheng (2015) proposed a method to measure performance for serious games by inserting Event Listeners and Event Tracers inside the game to track behavioral telemetry data. The Event Tracers collect data when the player triggers the Event Listeners throughout the game, capturing information related to timestamps, world position, and other game variables. However, this approach only gathers data when the player's avatar triggers an Event Listener when exploring the map. These Listeners need to be manually placed by designers in the game map and can be viewed as sampling since it only gathers data when they are triggered, losing all possible data executed between Listeners.

Another recent work developed by Kang *et al.* (2017) gathers gameplay data from player interactions to analyze behavioral patterns. That work focuses on the learning aspect of the player by comparing interaction data with the various game objects to solve puzzles. However, the gameplay data gathering is restricted to only capturing interactions with certain objects and is not broad enough for performing other types of analysis besides the one proposed by the authors.

4.5 FINAL CONSIDERATIONS

This chapter presented the concretization of PinG, a conceptual framework for game telemetry that tracks the actions and events alongside with their cause-and-effect relationships, through our concrete framework in Unity (PinGU). Our framework facilitates the process of tracking and storing provenance for data exploration and analysis. This provenance can aid the detection of gameplay issues, support developers for a better gameplay design, aid identifying game sections where players had issues and the reasons behind these issues. It can also be used for mining behavioral patterns from individual sessions or groups of sessions. We also showed the integration of PinGU in four different games with completely different styles to enable provenance tracking.

Our novel concrete framework, which in contrast to the original PinG, is no longer a conceptual framework and actually provides implementations to track provenance data through PinGU. All the data management and graph generation are done automatically and without user interference. Designers need to import PinGU inside the project, attach the provenance classes in the respective objects, and create domain-specific tracking methods in the desired events. The more complex process is related to influences, which designers need to know a priori the possible causal relationships in order to correctly track them.

However, tracking provenance data is only the first step in the analysis process. In the next chapter, we present our advances in the provenance visualization field through our visualization tool named *Prov Viewer*. *Prov Viewer* allows designers to perform exploratory analysis over the tracked provenance data through its interactive graph rendering. Our tool is fully compatible with PinGU and also with provenance data that is based on the PROV Model.

CHAPTER 5 – PROVENANCE VISUALIZATION

5.1 INTRODUCTION

Displaying game data is an issue in present times, bringing problems related to scalability when dealing with long game sessions or by having too many actors/players. Using provenance as a method of gathering game data only escalates this issue due to the richness and highly detailed data, generating huge quantities of data as a consequence. Although there are some tools in the literature for graph analysis (BASTIAN; HEYMANN; JACOMY, 2009; ELLSON et al., 2004; RIO; SILVA, 2007; SELTZER; MACKO, 2011), they are based on these simple node-link diagrams with only basic visualization features, such as labels and colors to distinguish edges and vertices, neighbor detection, and size for different intensities. However, using these simple node-link diagrams to represent provenance data can also harden the graph understanding when dealing with the wealth of information that can be contained in a single provenance node, even when using the different shapes to distinguish the information.

Aside from graph visualization tools, there are also some scientific workflow systems that provide proprietary methods of displaying provenance data as well as standalone tools. Nevertheless, workflow management systems normally lack graph manipulation features for viewing provenance graphs and offers basic static views for the provenance data. On the other hand, the standalone tools offer limited interactive features but they were built for specific domains.

Thus, we initially experimented with new methods to visualize provenance in the context of games (KOHWALTER; CLUA; MURTA, 2013). We developed an initial prototype for provenance visualization and used it on the SDM (KOHWALTER; CLUA; MURTA, 2011) game to assess whether provenance data visualization can be helpful in the understanding of game events (KOHWALTER; CLUA; MURTA, 2014). This prototype was also used in another application that extracts game provenance through image processing mechanisms (JACOB et al., 2014). Along the road, we decoupled our prototype from the SDM game and generalized it, thus allowing its use for general provenance visualization.

In this chapter, we introduce *Prov Viewer*, a graph-based visualization tool that we developed for interactive exploration of provenance data that is compatible with the PROV

model. *Prov Viewer*[11] processes the collected provenance data to generate an interactive provenance graph to provide advanced visualization features for identifying steps and contributors to a given result.

The tool we present in this chapter is the result of several extensions and new techniques we developed to address issues encountered in different scenarios. We have designed new visual representations and interaction mechanisms that address many of the aforementioned challenges: (1) domain configuration, customizing the visualization for specific needs; (2) interoperability, supporting PROV-N for importing provenance data; (3) shapes, sizes, and colors, supporting a clear distinction of information types; (4) collapsing, highlighting the relevant information in the graph; (5) filtering, removing information that is not relevant for a given analysis; (6) graph merge, integrating the analysis of multiple trials, and (7) specialized layouts, organizing the graph in a more understandable way and using the game map as a background.

This chapter is organized as follows: Section 5.2 details our provenance visualization tool, *Prov Viewer*. Section 5.3 presents three different case studies using our tool. Section 5.4 presents some of the related work in the area of provenance visualization. Lastly, Section 5.5 provides the final considerations for this chapter.

5.2 PROV VIEWER

In this section, we describe our visualization tool for displaying provenance graphs: *Prov Viewer* (Provenance Viewer). Our tool is compatible with the PROV-N notation, allowing its adoption in different domains and applications. The provenance data, which contains the provenance information among entities and their relationships, is processed to generate a provenance graph. This graph is a visual representation of the provenance data and supports user interaction, which is a key feature for understanding how each action influenced in the outcome and how they influenced each other. It is also possible to manipulate the graph by omitting facts and collapsing chains of actions for a better understanding and visualization experience. No information is lost in this process so that the user can undo any changes made during analysis.

[11] *Prov Viewer* is available at https://github.com/gems-uff/prov-viewer

Prov Viewer uses the PROV notation, where square vertices represent activities, circles represent entities, and pentagons represent agents. Furthermore, each vertex is composed of multiple attributes that describe the vertex. Each attribute contains a label and a value that is associated with it (*e.g.*, startTime: 2012-05-25T11:15:00, endTime: 2012-05-25T12:00:00). The edges in the provenance graph represent the relationships between vertices. As such, activity vertices may be positively or negatively influenced by other vertices and have relationships with entities and agents.

Before using *Prov Viewer*, it is necessary to configure it to understand the domain peculiarities and customize the visualization features. This is accomplished by creating a config.xml file based on the configuration schema of *Prov Viewer*. This configuration file allows the user to customize the graph visualization. *Prov Viewer* also has a feature for automatic detection and configuration for each edge type and color scheme, which represents most of the configuration effort of the tool, according to the graph being used. Note that the user needs to manually input specific parameters in the configuration XML file in order to use some of our tool layouts. However, this task is done only once for a new domain.

Figure 27 shows the high-level architecture of our tool, illustrating some of its features that allow users to interact with the provenance data and identify relevant actions that impacted the results. The following sub-sections describe the most relevant features, including shapes and colors to distinguish information, manual collapses, graph merges, layouts, and automatic collapse.

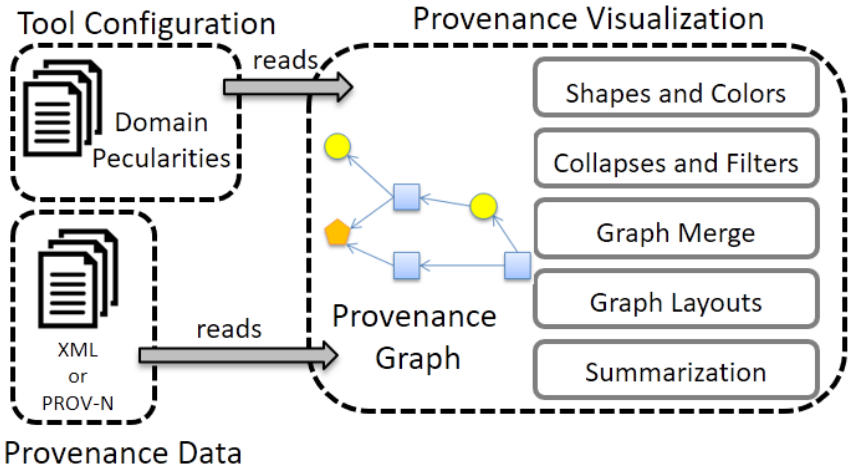


Figure 27: *Prov Viewer's* high-level architecture

5.2.1 **DATA FORMAT**

Aside from the configuration XML file, *Prov Viewer* expects the gathered provenance data extracted from an application to be formatted using a simple XML Schema (input.xsd[12]). *Prov Viewer* is also compatible with the PROV-N format but it loses some of its features due to format restrictions and limitations when exporting the graph after manipulating it inside our tool. We describe *Prov Viewer*'s XML format for provenance data since PROV-N is already documented at the PROV website. The input XML contains the list of vertices and edges that represent the provenance data. An example using the input schema is illustrated by Code 6.

```
<vertices>
  <vertex>
    <id>ag01</id>
    <type>Agent</type>
    <label>Joao</label>
    <date>0</date>
    <attributes>
      <attribute>
        <name>Job</name>
        <value>Senior</value>
      </attribute>
      <attribute>
        <name>Level</name>
        <value>1</value>
      </attribute>
    </attributes>
  </vertex>
  <vertex>
    <id>ac01</id>
    <type>Activity</type>
    <label>Action</label>
    <date>1</date>
    <attributes>
      <attribute>
        <name>Role</name>
        <value>Manager</value>
      </attribute>
      <attribute>
        <name>Task</name>
        <value>Aid</value>
      </attribute>
    </attributes>
  </vertex>
</vertices>
<edges>
  <edge>
    <id>e01</id>
    <type>Neutral</type>
    <label>wasAssociatedWith</label>
    <value>0</value>
    <sourceid>ac01</sourceid>
    <targetid>ag01</targetid>
  </edge>
</edges>
```

Code 6: Vertex and Edge lists

[12] https://raw.github.com/gems-uff/prov-viewer/master/src/main/resources/Graph/input.xsd

For both vertex and edge, the “id” is a unique identification name and “type” represents the vertex’s category (activity, entity, or agent) and for edges, it represents the name of the relationship. The “label” is a human-readable representation that can also be used for sub-typing. The “date” is related to temporal information and “attribute” is used to register vertex attributes aside from label and date that are accessed by color schemes. For the edge, “value” represents a numerical value associated with the relationship and is used to determine the edge’s color and thickness. Lastly, “sourceid” and “targetid” represent the relationship members. As the name implies, these vertices correspond to the source of the edge and its target. *Prov Viewer* uses this information to generate the graph’s vertices and edges.

As previously mentioned, the graph vertices represent provenance concepts which are classified in *Prov Viewer* as agents, activities, and entities. Although PROV allows sub-typing, *Prov Viewer* always treats provenance concepts via their respective super classes. However, *Prov Viewer* allows vertices of the same type to contain different characteristics (in terms of extra attributes) without generating conflicts in the graph. Each vertex type has established characteristics, as described earlier, such as color and shape, which are defined by their type. However, their extra attributes can be different even when they belong to the same type, with the exception of the only two fixed attributes that all vertices have: label and date.

5.2.2 **SHAPE AND COLORS**

Prov Viewer builds its visualization strategy based on shapes and colors, for both vertices and edges. Shapes are used to map semantic concepts from the provenance and colors are used to map scalar values, such as intensity or orientation. The vertex shape is directly related to provenance semantics (*i.e.*, agent, activity, entity), while the vertex color is used for mapping scalar values through the usage of a color scheme. When selecting the desired attribute, all vertices with the specified status have their colors changed according to their respective values. We adopt the traffic light scale (DIEHL, 2007), which indicates the status of the variable using gradients from three colors: red, yellow, and green. The resulting color is automatically inferred from the minimum and maximum values for that attribute or using boundaries manually specified by the user in the configuration XML file. Enabling this type of feature allows the user to easily identify situations where the desired attribute value fluctuates throughout the data.

Both the edge shape (*i.e.*, thickness) and its color are used to show the intensity of the relationship. The intensity is the value associated with the edge, if any, and is more common

on influences (*i.e.*, *wasInfluencedBy*). A thin edge with a darker color represents a low influence relationship (*i.e.*, the assigned value to the edge is low). On the other hand, thicker and brighter edges represent a strong or intense relationship. Figure 28.a shows an example of edges with different colors and thickness and Figure 28.b shows the vertex color based on their time values (also represented by columns). This feature can be used to quickly identify strong influences in the graph just by looking at the edge's thickness and brightness. The edge's color is also used to represent any additional numeric information contained in the relationship (*e.g.*, influences that has numeric data), which can be any of these three types: positive, which is represented as green and indicates an increase in the numeric value (*i.e.*, when the edge has an associated positive value); negative, which is represented as red and indicates a decrease; and neutral, which is represented as blue and indicates no numeric chances.

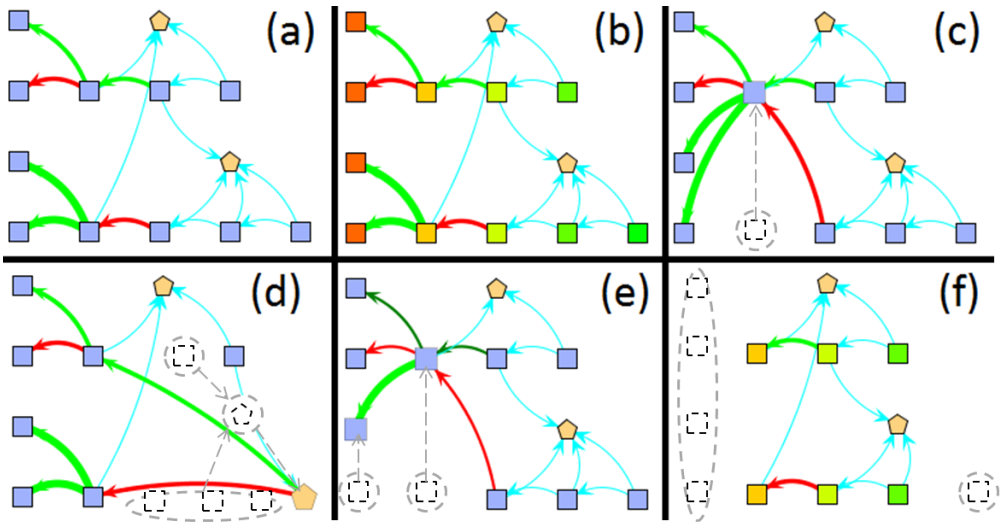


Figure 28: (*a*) Original graph; (*b*) graph with a color schema; (*c*) collapse of two activities; (*d*) collapsing of the agent's activities; (*e*) graph *c* after another collapse, and (*f*) temporal filter

5.2.3 **COLLAPSE AND FILTERS**

Our tool provides a vertex collapse feature to aid in the analysis of the graph and allows for a manual collapse of selected vertices in order to compact the graph size, grouping the selected vertices together in a single summarized vertex. No information is lost in this process and it can be reverted by the user. Figure 28.c shows an example of collapsing activity vertices. The grey markings represent vertices before the collapse and the grey arrow represents where they were collapsed to. Another usage of the collapse feature is to group activities related to the same agent, allowing the user to see all the influences and changes that the agent did throughout his tasks. Figure 28.d illustrates this example. The size of the collapsed vertex is bigger than the rest due to the number of vertices in the *collapse group* (and is proportional to the number

of vertices). Furthermore, the shape is the same as that of an agent vertex because there is an agent vertex in the *collapse group*.

The summarized information is displayed as follows: For String values, it shows all different values separated by a comma (*e.g., String_Value1*, *String_Value2*, *String_Value3*). For attributes with numeric values in a *collapse group* composed of two vertices, the tool shows the average value for that attribute followed by the minimum and maximum values. Otherwise (*collapse group* containing more than two vertices), the tool displays the average value followed by the five-number summary (minimum value, 1st quartile, median, 3rd quartile, maximum value).

Similar edges (*i.e.*, same type) that have the same target and type are also grouped together when collapsing vertices. The collapsed edge's information (*i.e.*, color, thickness, and value) is computed by summing or averaging the values of the participating edges, depending on their type. For example, *Prov Viewer* can use the *sum* function for edges representing expenses and the *average* function for edges representing a percentage. However, the user needs to parameterize each edge. Otherwise, the tool will always use the default *sum* function. Figure 28.e shows an example of collapsing edges that occurred when collapsing another group of vertices after collapsing the vertices in Figure 28.c. Note that the colors for each edge changed after the collapse due to the new maximum value (from the sum of the collapsed edges).

The tool also offers another way to filter vertices based on temporal information. The user defines the desired temporal range (*e.g.*, start time and end time) for visualization and the tool hides all vertices that are outside the selected range. Figure 28.f shows an example of the temporal filter, hiding vertices from Figure 28.a with time (which can be seen by the rows) greater than four and less than two. *Prov Viewer* also has an edge filter, which filters edges by context (*i.e.*, label) or by the type of relationship.

5.2.4 **GRAPH MERGE**

Our provenance visualization tool also proposes a feature based on Koop *et al* (2013) to merge two provenance graphs in order to generate a single unified provenance graph. The merge process combines the currently displayed graph with another graph, chosen by the user, to generate a single unified graph for visualization and storage. The merge process is composed of four steps: (1) vertex matching, which selects pairs of vertices from the graphs; (2) similarity verification between vertices, which receives two vertices from the first step and informs if they are similar. Vertices are considered similar if they belong to the same vertex type and have the

same properties with similar numeric values within a configurable margin of error; (3) merge of vertices that were considered similar in the previous step; and (4) creation of the unified graph for visualization, which only occurs after the matching process is over. The resulting graph can be exported using the PROV-N notation for future usage.

Figure 29 illustrates the graph merge of two distinct graphs from the same domain. Red vertices in the merged graph (Figure 29.c) belong exclusively to the first graph (Figure 29.a), while grey vertices represent common vertices (*i.e.*, merged vertices) from both graphs, and green vertices belong exclusively to the second graph (Figure 29.b). This graph merge feature is useful when analyzing multiple sessions or trials by detecting common sections. Merged vertices from this feature also provide similar summarized information using the five-number summary.

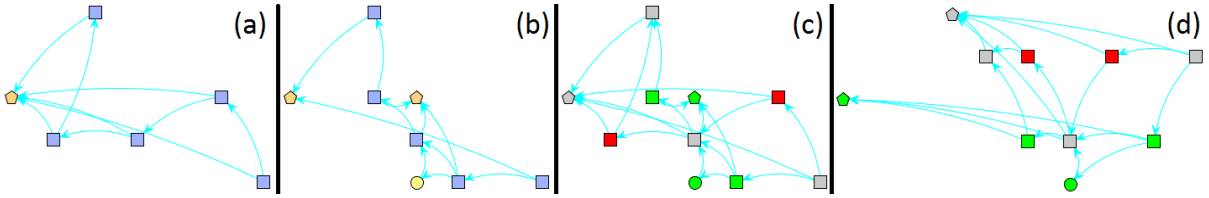


Figure 29: Two graphs (*a* and *b*) merged into a single graph (*c*) and with a temporal layout (*d*)

5.2.5 **GRAPH LAYOUTS**

Our visualization tool allows the user to interactively change the graph layout to better visualize the result. We created two provenance graph visualization layouts: temporal and spatial.

The **temporal layout** organizes the graph in a chronological order similar to a timeline (or spreadsheet) for each agent. Thus, each timeline (or line) of the graph groups activities of the same agent and each column in the graph represents the passage of time. This makes easier to know the entity or agent responsible for executing each activity by just looking at the agent responsible for that line. Thus, the graph positions the vertices in the *x* axis according to the chosen scale. Figure 29.d illustrates an example of our temporal layout, displaying the graph similarly to a spreadsheet and organizing the vertices in their chronological order. Note that now it is much easier to identify the agent responsible for each activity as the leftmost nodes and their chronological order by looking at the activity's placement in the graph. The horizontal position represents the time axis (passage of time) and the vertical position represents the agent axis (the responsible for the activity).

The **spatial layout** organizes the vertices in the graph by their spatial coordinates and can be used for spatial or georeference the data. This is particularly useful for corresponding elements with other graphical representations, such as a map of a city or a game scenario. When using the spatial layout in conjunction with a background image, the user sees where each event occurred just by looking at the graph's placement in the image. This layout also supports the usage of an orthographic image or maps taken from *Google Maps* and *OpenStreetMaps* as background for the graph. When dealing with real world maps, *Prov Viewer* automatically transforms the latitude and longitude to pixel coordinates.

5.3 CASE STUDIES

Prov Viewer was initially designed for the digital games domain. In the game domain, our visualization tool was used for the analysis of game sessions of five different games (*SDM*, *Super Mario World*, *Unity's Tower Defense*, *Unity's Angry Bots*, and *Unity's Car Tutorial*). We will cover only two of these five case studies of our visualization tool. However, since we decided to generalize our visualization tool, we also present an additional case study in another domain just to demonstrate that *Prov Viewer* is generic enough for general provenance viewing. This last case study is from the urban data domain, which we used our tool to analyze bus traffic data from the city of Rio de Janeiro. In the following sub-sections, we present all these four case studies of our visualization tool.

5.3.1 **CAR TUTORIAL**

The first case study is the Car Tutorial from Unity asset store, previously discussed in Chapter 4. We can use the car's coordinates on the track to plot the graph so that it is possible to visualize where the player was when the action was executed. This visualization also allows the designer to quickly identify which sections of the track the player had trouble. Thus, we can take advantage of spatial-referencing the data during the provenance visualization. We used a screenshot of the game map taken by our camera module with resolution of 1070x802.

Figure 30 shows the provenance graph of one game session, using the car's coordinates and the track's picture as background. This graph is composed of 169 vertices and 867 edges extracted from a 107-second game session, which represents one complete lap in the track. The vertices are colored according to the car's speed (gradient from white when close to zero and green for high values) and the visible edges are the speed delta between vertices.

Figure 30: Spatial referencing provenance data using a vertex-coloring schema according to the car speed.

We can quickly identify sections of the track that the player may have had issues, either by reducing the speed too much or by crashing, by just looking at the plotted graph in the race track. As an example, Figure 31 shows a zoomed section of the graph to better illustrate the reasons behind a car crash. The zoomed section of the graph has a different vertex-coloring scheme to differentiate events. By analyzing it, we can see that the car crash (red vertex) was influenced by two factors. The first one was on the previous curve, where the car lost contact with the ground (purple vertex with a blue edge linking the crash) after passing through the rumble strips at the end of the maneuver, thus preventing the player to prepare for the following turn. The second reason was that the player was too fast, as indicated by the red edge from the blue vertex, which is a reduction of the car's turn rate due to high speed.

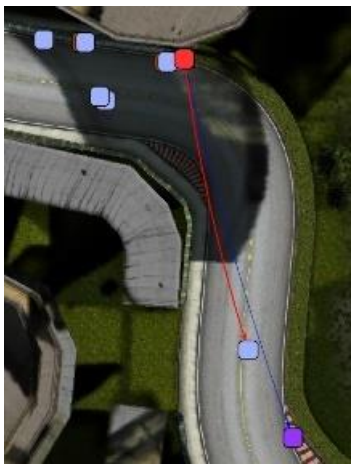


Figure 31: Influences behind a car crash showed using a different coloring schema to differentiate events.

Using the provenance from other laps of the race, we can begin to detect patterns during the game session or even compare the player's performance between laps. This analysis can also be extended to different game sessions by comparing the generated provenance graphs. Figure 32 illustrates an example of the generated provenance graph when gathering data from multiple laps during a single play session, enabling the designer to detect behavioral patterns and locations where the players are struggling the most. For example, Figure 33 shows a section of the track that is characterized by having multiple curves in the track. We can see the player's performance during each lap of the race, where each lap is represented by a different edge color. The first, second, and third laps are presented by red, green, and blue edges, respectively. Moreover, the first and last vertices of each lap are marked with circles of the same color as the edge and the timestamps are represented by the yellow numbers beside the vertex. As we can see, the player had approximately the same speed in all laps due to the same shade of green when entering this section of the track. However, the player took 15 seconds to pass through this section of the track on the first lap (52 - 37), 17 seconds during the second lap (131 – 114), and 10 seconds on the third lap (200 – 190).

Figure 32: Provenance graph from multiple laps.

Figure 33: Zoomed section from yellow rectangle on Figure 32.

By analyzing Figure 33, we can see a purple edge connecting vertices 47 to 49 that represented the reason behind the crash in the first lap (marked by the purple circle). This purple edge represents a cause-and-effect relationship, showing that the crash happened because the player passed through rumble strips (brown circle) and, as a result, lost car stability, could not complete the turn. Furthermore, notice the steep angles the player had to make due to his positioning in each curve. During the second lap (green edges), the player tried to avoid the crash by reducing speed. However, the player reduced too much speed to enter the second curve (white-green vertices). During the third lap (blue edges), the player managed to improve his performance and avoid any crashes by better positioning the car before each curve and thus reducing the necessary angle to make the turn while maintaining a nearly constant speed.

5.3.2 **ANGRY BOTS**

We conducted a second case study using a very different style of game, called Angry Bots, also from the Unity asset store, as presented in Chapter 4. In the available scenario, the player must face enemy robots and interact with the environment in order to complete the level. Figure 34 illustrates one of the possible visualizations of the provenance data gathered from an Angry Bots session, showing the vertex visualization scheme for the player's health attribute value (vertex color using a traffic light scheme) and the edges that influence on it (green and red edges) as the game progresses. Blue vertices represent other characters in the game (enemies), blue edges represent the chronological order of events, and green edges represent player’s health generation due to his passive regeneration ability. By analyzing Figure 34, we

can see the chronology of events, regions visited by the player, sections where more action happened, places where the player engaged in battle, and when the player suffered heavy health loss.

Figure 34: Picture of the entire graph. Vertex coloring based on player's Health attribute.

Considering that the player recovers health periodically, it is possible to infer that the cause of some deaths was the rush through the level without waiting to recover health or because of a tough enemy. Figure 35 illustrates the first case, where the player tried to rush through the game without waiting to regenerate the player's health, lost from previous battles. The dashed blue arrows were added to the figure to highlight the player's general movement and do not belong to the provenance data.

Figure 35: Player's health when trying to rush the game.

Figure 36: Sequence of events of the player exploring a section of the map and confronting an enemy.

After the player engaged an enemy in a major battle, which the player didn't leave unscathed by looking at the orange vertices, the player continued advancing through the level.

Then, on the player's third major engagement, where the player was still wounded by looking at the orange vertices, the player lost most of the remaining health, as illustrated by the following red vertices. Even though the player was low on health, he managed to dispatch his enemies on the forth battle without losing a single health point (no red edges). However, the player continued pressing on without resting, which would allow for him to gradually restore the lost health points before the next engagement, until dying on the next battle when the player got hit by the enemy (Battle #5).

Figure 37, Figure 38, and Figure 39 illustrates the second case, showing the sequence of events that led the player to a tough engagement (Figure 39). By analyzing the picture, we can see that the player started these events with good health (green vertices in Figure 36), leaving the first battle slightly injured (yellow vertex). A few moments later he encountered another enemy in a side room (Figure 37), where once again he overcame the enemy with only minor wounds (the vertex is still yellow). However, just when he left the room, the player was ambushed by another enemy that was patrolling the corridor (the new blue vertices in the corridor from Figure 38). This enemy was a *mech*, which is much tougher than a regular enemy (notice the high number of dark red edges that represent player doing damage to the enemy). This battle resulted in the player's death after getting hit by two rockets (Figure 39) followed by his resurrection shortly after (green vertex in the bottom of Figure 38 that is linking the green edge to a red vertex).

Figure 37: Continuation of the sequence of events from Figure 36 with the second engagement inside a room.

Figure 40 illustrates the moments when the player died, which are marked by red circles. Meanwhile, the orange circles illustrate the player "refreshed" state after resurrecting,

Figure 38: Continuation of the events from Figure 37 that led the player to a tough confrontation that resulted in his death.

Figure 39: A zoomed section from Figure 38 showing both the moments the player was hit by the enemy's rockets. Filtered to show only the edges that affected the player's Health.

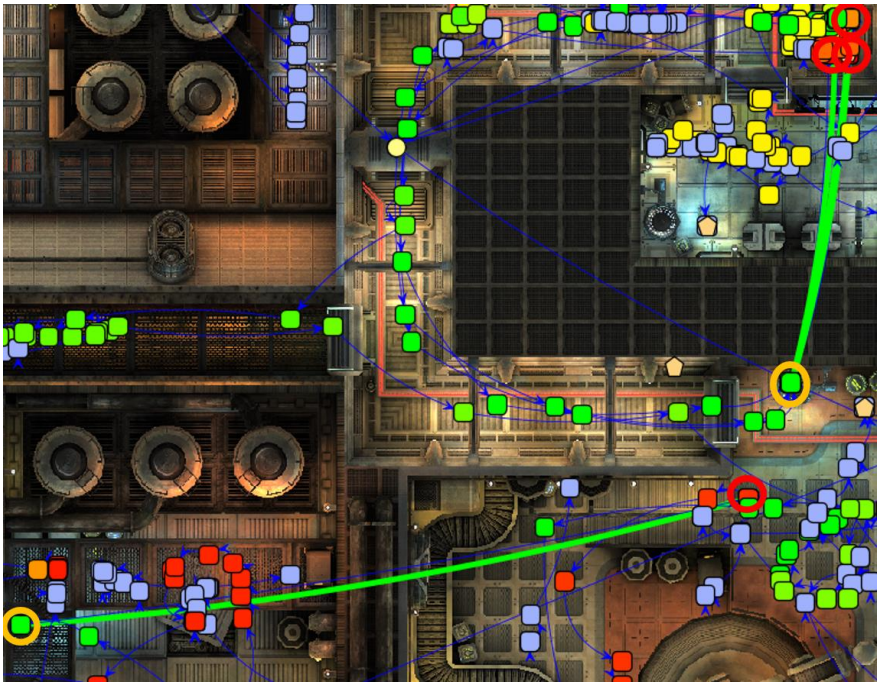


Figure 40: A filtered graph showing the moments the Player died and was resurrected.

as well as the resurrected location. Both situations have a green edge linking the player's death to the resurrection, which shows that his health went from zero (red vertex) to maximum (green vertex) after resurrecting. Notice that the player died three times trying to beat the *mech* enemy from Figure 38 before finally defeating it.

5.3.3 **BUS TRAFFIC**

The third case study is based on bus traffic data analysis in the city of *Rio de Janeiro*, which we use to show that *Prov Viewer* is able to handle provenance outside the gaming domain. The data used in this research, which includes geographic location tracked from the buses' GPS, are obtained from public JSON files at *DataRio*[13]. *Prov Viewer* was used in this context to render the data for analysis, allowing the research team to understand the wealth of tracked information. Our tool allows for filtering the data to focus on specific buses or relate the bus delays with ongoing events in the city through their geographic location, speed in the region, and timestamp.

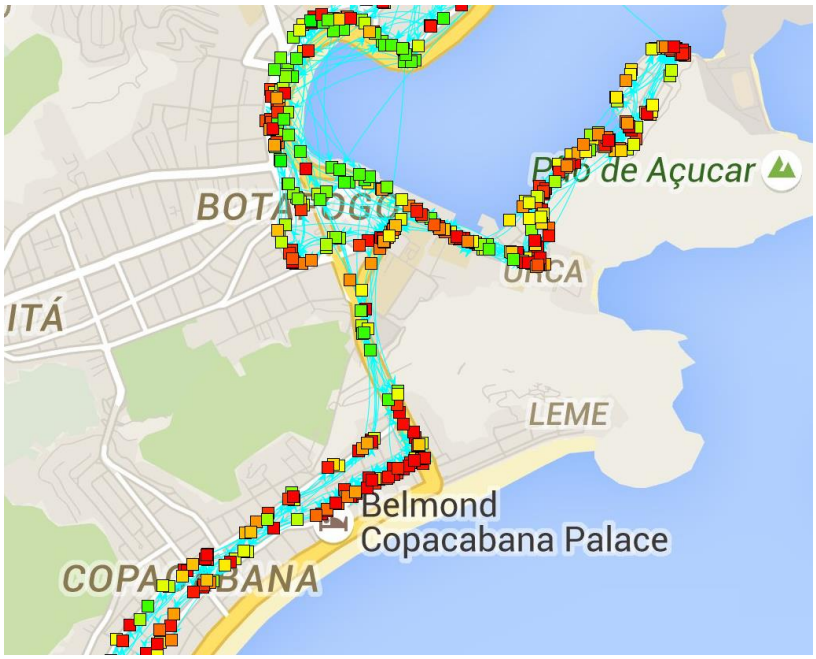


Figure 41: Provenance Graph rendered on *Prov Viewer* from collected bus traffic data over a partial map of Rio de Janeiro city from Google Maps.

Figure 41 illustrates one of the possible visualizations of the provenance data using *Prov Viewer*. The graph contains 601 vertices and 600 edges. The displayed graph uses a color schema based on the bus speed. Therefore, the vertices, which represent on-line GPS information tracked from buses at every minute interval, are colored from red to green according to their instantaneous speed, while the blue edges link these vertices in a chronological order. Note that the displayed graph is showing bus data from nine different buses from the same route within a period of two hours.

---

[13] *DataRio*: http://data.rio/dataset

In the graph from Figure 41, we can see the buses routes through the city and their respective speeds along the way. Furthermore, we can see that the traffic is better in the region near "*Botafogo*" due to the high concentration of green vertices than "*Urca*" and "*Copacabana*". This type of graph visualization allows the user to quickly identify the streets where the buses moved slower due to traffic by finding regions in the graph with reddish vertices. Moreover, it is also possible to better understand the extension of the traffic jam and the affected areas by crossing the displayed graph with a graph from another route (*e.g.*, merging the graphs) that also use segments of the same street. Furthermore, it is possible to analyze each bus separately and their progression in the map by using the visualization features presented in this chapter and previously shown in the other case studies.

5.4 RELATED WORK

The related work can be grouped into two categories: workflow management systems that have built-in provenance visualizations and standalone provenance visualization tools. The workflow management systems that have built-in provenance visualizations (ALTINTAS et al., 2004; CALLAHAN et al., 2006; HULL et al., 2006) allow for easy integration between provenance collection and analysis. However, they have a shortcoming of not supporting provenance data generated by other workflow management systems or standalone provenance gathering tools, even when they are compatible with well-known provenance models. Furthermore, workflow management systems normally lack graph manipulation features for viewing provenance graphs.

On the other hand, standalone provenance visualization tools are closely related to our work. Provenance Explorer (CHEUNG; HUNTER, 2006) takes RDF-based provenance outputs from gathering systems and dynamically generates customized views of provenance trail. However, it focuses on provenance data and inference rules associated with processing events in a laboratory or manufacturing plant, lacking the support for data processing activities in the digital domain. Furthermore, their collapse feature only supports one expansion level, instead of multiple levels of detail.

The ZOOM (BITON et al., 2008) prototype provide users with an interface to query provenance information generated by a workflow system through SQL queries. An interesting aspect is that it allows the user to dynamically modify the graph by hiding irrelevant information, updating the provenance graph for the new view.

PROV Toolbox is another existing approach, which converts W3C PROV data model representations. However, it lacks a built-in visualization and requires the use of a generic graph tool (*Graphviz*) to visualize the provenance data.

Another similar tool is PROV Translator, which validates PROV representations and translates them to other representations. It also provides graph visualizations based on their previous work (EBDEN et al., 2012), which displays a provenance graph using PROV's vertex shape and color to identity the vertex type.

The PROV-O-Viz (HOEKSTRA; GROTH, 2014) tool is a web-based visualization tool for provenance based on PROV that uses *Sankey Diagrams* for visualization. *Sankey Diagrams* are used to visualize flow magnitude between nodes in a network and in PROV-O-Viz the activity or entity width is based on the information flow.

Some related work is limited to specific domains (*i.e.*, Provenance Explorer), require additional knowledge (*i.e.*, ZOOM, PROV-O-VIZ), or are not compatible with provenance data from other tools (*i.e.*, Kepler, VisTrails, Taverna). Nevertheless, they individually provide some interesting features, such as interactive graphs, level of detail, summary nodes, merges, and filters. However, these approaches do not provide these features in an integrated way, hindering the analysis due to visualization and manipulation restrictions, which sometimes require additional external procedures. Moreover, they lack any means of overlaying provenance information onto a spatial structure for analysis.

5.5 FINAL CONSIDERATIONS

Graph visualization strategies bring problems related to scalability when dealing with provenance datasets beyond a few hundred nodes, which is common in games. Traditional node-link diagrams to represent game sessions can easily become too visually cluttered when dealing with huge quantities of data, limiting the user’s ability to thoroughly analyze and explore the data. To deal with this problem, *Prov Viewer* offers collapse options that can generate different levels of detail and graph layouts to sort the data and reduce node clustering. *Prov Viewer* also has some basic automatic collapses based on vertex similarity and graph merges, allowing users to omit data and combine different graphs for analysis. Other contributions include the merging of PROV-N data from different files and georeferencing capabilities for provenance information.

Our tool can be configured and used by different provenance applications as a general-purpose provenance visualization tool since it supports graphs that use the PROV-N notation.

Prov Viewer also supports pre-processing steps, which can be done outside the tool, as long as the final data format is compatible with PROV-N or the tool's unique notation. We showed four case studies from different domains: three analysis of digital game sessions and one analysis of bus traffic data.

Game sessions can generate huge graphs depending on the number of actions executed or how long it lasted. This huge quantity of data can provide visualization challenges due to the overwhelming size of information, impacting in the ability to understand and analyze the contents of the graph. Thus, in the next chapter, we present three different summarization techniques based on clustering that can reduce the provenance graph by hiding unimportant data. We also provide two detailed evaluation of our summarization algorithms.

CHAPTER 6 – PROVENANCE SUMMARIZATION

6.1 INTRODUCTION

Depending on the usage context, the amount of tracked game data can reach huge sizes, affecting negatively the capability of analyzing game data and often requires processing the collected game session data. Depending on the game style, a single game session might take several hours to be completed. This results in an overwhelming quantity of displayed data, making the analysis difficult for the developer due to the large volume of tracked information.

The existing approaches for game telemetry use information clustering as a reducing technique to deal with visualizations of large quantities of tracked game data. Common clustering techniques involve density clustering (SANDER et al., 1998), which create clusters based on the information distribution in a region, and hierarchy clustering (SIBSON, 1973), which uses the notion of proximity through a distance metric and a distance function. However, both methods consider spatial information for clustering, ignoring temporal relationships when summarizing data. Thus, the sequence of events is lost in the process, favoring the perspective of information distribution in the game scene.

Therefore, in this chapter we propose three collapse algorithms based on DBSCAN (ESTER et al., 1996), a popular density clustering algorithm, to summarize tracked telemetry data that considers the sequence of events instead of their spatial information. This helps to manage the volume of data and lets the users focus on events that may be more important than others. These collapses reduce the volume of displayed information by hiding part of the data that were not significant enough to produce a meaningful impact on the game and did not offer any useful information for analysis. As the proposed collapses preserve the sequence of events and consider temporal information instead of using spatial information to group nearby vertices, designers can still track the player's progress in the game and easily identify noteworthy regions that had a meaningful impact in the game. While our approach (KOHWALTER; MURTA; CLUA, 2018) focuses on game analysis, we believe that our solution may be useful to many other systems that collect big data and relationships among them.

The overall goal of this chapter is to answer the following research question in relation to the three proposed algorithms and the existing DBSCAN algorithm:

RQ: Which type of similarity summarization is an effective method for reducing the information to be analyzed?

We implemented our approach in the open-source tool *Prov Viewer* (KOHWALTER; CLUA; MURTA, 2013), which displays game telemetry data collected with the PinG (KOHWALTER; CLUA; MURTA, 2012) framework. *Prov Viewer* shows the collected telemetry data as a provenance graph (MOREAU; MISSIER, 2010), where vertices represent the events and edges represent the chronological order in which these events were executed. We evaluated our algorithms within two experiments: (1) an automatic experiment to obtain quantitative results and (2) a manual experiment involving human judges to obtain qualitative results.

The rest of the chapter is organized as follows. Section 6.2 presents our approaches for hiding irrelevant information through collapses. Section 6.3 describes our evaluation methodology. Section 6.4 presents our automated evaluation, while Section 6.5 presents an evaluation using experts in the field. Section 6.6 presents the related work in the area of game session analysis, along with their clustering techniques. Finally, Section 6.7 provides final considerations of this chapter.

6.2 SIMILARITY COLLAPSE

As previously discussed, game sessions may generate large quantities of data, making it difficult for designers to visually explore their telemetry results. However, as multiple consecutive telemetry data actually represent subtle variations of the game state, they can be collapsed, keeping only the most relevant data. These remaining telemetry data actually represent relevant variations in the game state according to specified game attribute.

Our approach aims at reducing the volume of displayed information by summarizing the tracked data. In order to accomplish the summarization and to respect the sequence of events, the raw data must be structured as a graph, with vertices representing the events and edges representing the chronological order in which these events were executed.

Figure 42 illustrates an example from *Angry Bots* where the player experienced two distinct battles against different enemies, with each battle being represented by a yellow box. The blue edges represent the chronological order of events, while the red edges represent moments when an enemy hit the player. Vertex positioning also represents the timeframe (from left to right) and vertices within the same column mean that they occurred at the same time slice (*e.g.*, within the same 1-second window when using seconds as the time scale for visualization). Each vertex in the graph represents the execution of an action. The vertex color is proportional to health value, which ranges from green (high health) to red (low health). In the first battle,

there are zero red edges connecting the player's vertices. Therefore, the player managed to dispatch the enemies without taking any hit. However, in the second battle, the player struggled to overcome his enemy and lost a large amount of health, which is reflected by the vertices colors.

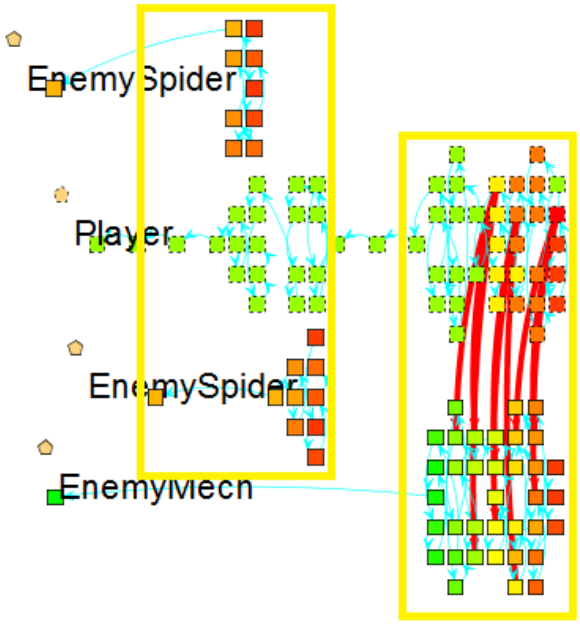


Figure 42: Two battle examples, each marked by a yellow box

If we are interested in analyzing the player's challenges, then we should focus on sections where the player struggled to overcome or had significant changes in the game state. As such, all instances that the player's hit points did not change or barely fluctuated are not relevant to the analysis, which happens to be the case of the player's first battle. Therefore, we can omit all vertices in the graph that have similar values with their neighbors by doing a collapse based on similarity. In the example illustrated at Figure 42, it would mean collapsing the first fight entirely and grouping some of the vertices from the second fight that shares the same color, resulting in a graph similar to the one illustrated by Figure 43.

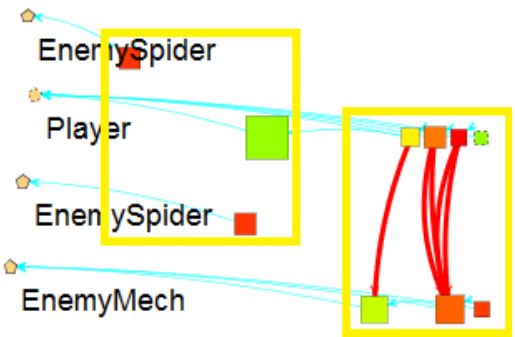


Figure 43: Graph from Figure 42 after the similarity collapse.

In order to achieve the graph from Figure 43, it is necessary to compare each vertex with its neighbor, which is connected by an edge, to omit similar states. If the vertices' values are similar for the specific game attribute being analyzed, they can be collapsed into a single vertex. Since the objective is to omit all similar states, it is necessary to go beyond the vertex first neighbor. If any of the vertices in the *collapse group* have an edge to a vertex outside of the *collapse group* and that vertex has a similar attribute value to those in the group, then it is added to the *collapse group*. Thus, the *collapse group* will keep growing until a significant change of state is detected.

We initially considered using Dimension Reduction algorithms (FISHER, 1936; F.R.S, 1901) to achieve this type of information reduction. However, Dimension Reduction works by removing the data and can lead to information loss if not handled with care, which results in a loss of flexibility when manipulating and examining the game data during visual exploratory analyses. Thereby, we decided to use density clustering algorithms to achieve this information reduction while at the same time maintaining the flexibility to manipulate and handle the data since the information is only clustered and not removed. Thus, the analyst can easily expand a cluster to explore the collapsed data. Furthermore, clustering algorithms can be used to create different levels of detail through successive clustering of the game data, enabling different overviews of the tracked data.

Similarly, we also considered using spatio-temporal clustering algorithms as a basis of our heuristics. However, there is no significant difference between density-clustering algorithms to spatio-temporal clustering algorithms since the spatial and temporal variables can be easily inserted in the distance function. This is basically what the ST-DBSCAN (BIRANT; KUT, 2007) and other similar spatio-temporal algorithms do.

We propose the usage of **DBSCAN clustering algorithm** as a basis to achieve this type of collapse, using the distance function to determine the similarity between vertices. Since provenance graphs are commonly composed of vertices with two or three neighbors, we had to discard the density parameter in order to use this type of clustering algorithm. In the DBSCAN algorithm, the similarity between values is then defined by the epsilon (ε) parameter, which is used by the distance function (*e.g.*, Euclidian distance). This epsilon must be related to the attribute (or attributes) being used for analysis and be an absolute value. Furthermore, the DBSCAN algorithm decides if neighboring vertices should belong to the same *collapse group* by comparing their distances (*e.g.*, values) with the epsilon. Note that a vertex is only a neighbor of the *collapse group* if it has an edge connecting it to the group. As such, the *collapse group*

only expands through the edges that connect the vertices in the graph and, therefore, the temporal sequence of events is always preserved because a *collapse group* can never be a disconnected sub-graph.

When taking out the density parameter from DBSCAN, the clustering algorithm could face situations where it would collapse the entire graph in a single node when vertices had small and gradual variations in their values (*e.g.*, crescent graphs, monotonic graphs). To mitigate this problem, we propose an adaption for the DBSCAN, named as ***IC* (inter-cluster verification)**, to extend the heuristic that defines the *collapse groups*. Instead of comparing only with the closest neighbor (in the case of graphs, the direct neighbor), the comparison would also be made with the entire group due to small variations that each vertex can have in its value without causing a change of state. Thus, the epsilon must be checked against the group's minimum and maximum values before adding new vertices. If adding new neighbor results in a difference between values greater than the epsilon, then it is marked as a change of state and the vertex is not added to the group, possibly implying in the creation of a new group with this new vertex and the similar ones. This modification limits the cluster growth through a distance restriction, imposing a variance limitation on the range of possible values by always comparing with the farthest member of the cluster instead of only the direct neighbor. Therefore, this heuristic avoids creating a single cluster when the data contains monotonic behavior.

To better formalize this process, consider $G = (V,E)$ as a directed graph where $V = \{v_1, v_2, ..., v_n\} \mid v_i, 1 \leq i \leq n$, represents an event related to an element of the game (*e.g.*, player or enemy) and $E = \{e_1, e_2, ..., e_m\} \mid e_j = (u, v), 1 \leq j \leq m, u \in V$, and $v \in V$, represents the influence of an event into another, which indirectly indicates the chronological order of events in the game. Furthermore, consider that $A = \{a_1, a_2, ..., a_k\} \mid a_l, 1 \leq l \leq k$, represents a numeric attribute used for analysis of each vertex. Moreover, consider S to be a similarity collapse set of connected vertices for a specific attribute and $min(S, a)$ and $max(S, a)$ as the minimum and maximum values for the attribute $a \in A$ considering all vertices in the *collapse group S*. A new vertex $x \in V$ is added to a *collapse group S* if and only if $abs(max(S \cup \{x\}, a) - min(S \cup \{x\}, a)) < \varepsilon$. This change is made only in the distance function inside the neighborhood query, which is responsible for returning all reachable points within the epsilon restriction, according to the *DBSCAN* algorithm. Thus, instead of comparing the distance with only the direct neighbor, the *IC* variant of the *DBSCAN* algorithm considers all the vertices already in the *collapse group*.

For example, consider $G = (V,E)$ where $V = \{v_1, v_2, v_3, v_4, v_5\}$ and $E = \{(v_5, v_4), (v_4, v_3), (v_3, v_2), (v_2, v_1)\}$. All vertices have only one attribute with the same value, but v_3 that has the

double of the value of the other vertices. Let us define the epsilon to be one standard deviation of the v_i values. By running the IC variant, it would return three disjoint sets: $S_1 = \{v_1, v_2\}$, $S_2 = \{v_3\}$, and $S_3 = \{v_4, v_5\}$. The reasoning for this is: first, it adds v_1 then v_2 to S_1. When trying to add v_3 to S_1, the difference between values is greater than the established threshold. Thus, v_3 is not added to S_1. Since v_2 has no incoming neighbor beside v_3, no other vertex is added to S_1. The same happens when evaluating v_3 against v_4 and thus only v_3 belongs to S_2. When starting with v_4, the algorithm inserts v_4 in S_3 and evaluates v_5. Since the difference between v_5 and v_4 is lesser than the epsilon, v_5 is added to S_3. Because v_5 has no other neighbor, the algorithm ends and returns S_1, S_2, and S_3. By using the proposed modification, nearby vertices in the graph that have similar values are collapsed into a single vertex, reducing the graph's overall size by creating clusters of similar information.

However, the *IC* variant might not be useful when the data distribution is more erratic but controlled in some sort, containing sections of very close vertices (e.g., 1.01, 1.0095, 1.015, 1.02) and others that are more distant from each other but still orbit around some value (e.g., 20, 24, 19, 22). Thus, we also propose the **VE (Variable Epsilon)** adaptation to be incorporated in the DBSCAN: instead of using a single universal epsilon to define the collapse sets, each set defines its own epsilon and then adapt it as the set grows in size. Thus, each set initially uses the universal epsilon until it reaches a certain size. After reaching the size threshold, each set computes its own epsilon based on the current members of the set. This set epsilon is then used for further insertions in its respective set and each set have their own epsilon based on their members. If a new element is inserted in the set, then that set's epsilon is recalculated based on its members, including the newest one. This allows for each set to only have members that orbit around the cluster's defined epsilon and allow each cluster to have its own defining characteristic that can be completely different from another cluster. However, since the epsilon can change as the set grows, then in a way it would be adapting accordingly to its surroundings, providing a controlled variation of its own epsilon. For the first prototype, we based the epsilon on standards deviations of the values from all members of the collapse set, including the new additions, which will provoke slight variations in the epsilon. However, the bigger the set, the lesser impact the new additions will have in the set's epsilon.

These two new variants described above (*IC* and *VE*) can also be incorporated together in the *DBSCAN*, generating three different variants: (1) inter-cluster verification with a fixed epsilon (IC), (2) no inter-cluster verification with a variable epsilon (VE), and the combination of both, resulting in (3) inter-cluster verification with a variable epsilon (ICVE). The **ICVE**

(Inter-Cluster verification with a Variable Epsilon) adaptation provides, even more, cluster growth control since new members need to be within epsilon distance of the farthest member of the cluster. However, it also allows for each cluster to have its own characteristic, creating a more knit group of likeness. Furthermore, we defined the initial epsilon to be based on the standard deviation of the attribute values from the graph being analyzed. Similarly, the algorithms with variable epsilon also updated their epsilon values based on the standard deviation of the cluster. Figure 44 shows a simple example of the different clusters generated over a linear graph by each of the presented variants (*i.e.*, IC, VE, ICVE).

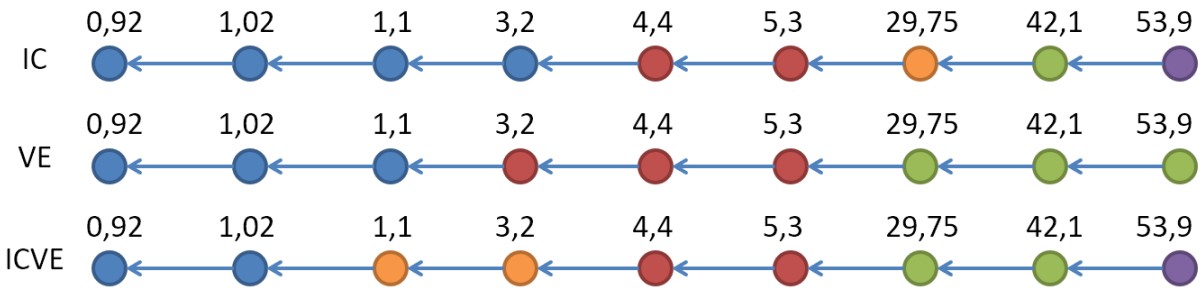


Figure 44: Example of the three clustering variants in action.

6.3 EVALUATION METHODOLOGY

In this section, we assess the proposed clustering algorithms for provenance graphs and determine which is the most appropriate algorithm to meet our goals. The algorithms try to detect similar sequences and summarize them into a single collapse, shrinking the graph to show only vertices that are semantically different from each other. Each adaptation proposed in this chapter accomplish this feat in different ways.

We elaborated two distinct experiments in order to answer our research question (RQ: Which type of similarity summarization is an effective method for reducing the information to be analyzed?): (1) an automated experiment and (2) an experiment with human judges. The experiment with human judges uses graphs from the automatic experiment and real provenance data from a single game session. These two experiments assess the usage of the DBSCAN algorithm and its three variations to summarize provenance data. In both experiments, we are evaluating the effectiveness of all three clustering algorithms in terms of graph reduction and semantic preservation using the DBSCAN algorithm as a baseline. We used a synthetic oracle in the first experiment to automatically evaluate each algorithm through precision, recall, and F-measure (RIJSBERGEN, 1979) metrics. The F-measure is the harmonic mean of the precision and recall metrics. A synthetic oracle is an artificially created oracle for generic graph

clustering. This allows us to measure the effectiveness of each algorithm in a broader aspect, using thousands of graphs for the evaluation.

However, due to the synthetic nature of the experiment and the very individual nature of the definition of "what is the best summarization", we also devised a second experiment that uses a smaller set of graphs to evaluate the four algorithms. Unlike the first experiment, which uses a synthetic oracle, the second experiment uses human judges to decide which algorithm is the most appropriate for each situation. However, we need to use a smaller set of graphs due to time constraints and to avoid burnout of the judges. Nonetheless, in both experiments, we evaluated different types of provenance graphs and two different numeric behaviors: (1) random noise and (2) monotonic noise.

In a brief study, we narrowed down the types of graphs that provenance information can be produced based on different game genres into three categories: (1) Directed Acyclic Graphs (DAG), (2) Tree Graphs, and (3) Linear Graphs. DAG is the most common form of provenance graph. Role-playing games, action, racing, sports games, and any other game with multiple actors will generate a DAG graph. However, provenance graphs can also result in a tree-graph under special occasions. For example, a provenance graph can be a tree when the information is still being gathered or when analyzing the provenance data from multiple sessions at the same time to know how and where each player experience diverged. Lastly, the provenance graph can be a linear-graph when only one agent/actor is involved and past actions have no influence on the current action, besides the most recent one. Games that might generate a linear-graph are puzzle games. Linear graphs can also be generated when there are multiple agents but they do not interact with other agents. When this occurs, the resulting graph will contain multiple disconnected linear graphs. Lastly, analyzing game data strictly from a single actor might generate a linear graph.

Each graph category was based on a template (a graph of the clustering result) and different graphs were generated through the addition of noises in order to verify if the algorithm would correctly omit these added elements through collapses. These templates were based on already known graphs for each category being evaluated in this experiment and represent the most common graph structures that can be achieved from real tracked provenance data from games. The template vertex values were randomized, following a uniform distribution with integer values ranging from negative to positive values, thus simplifying mathematical calculations and minimizing numerical precision errors. Moreover, each template vertex also represents a collapse resulting from the ideal clustering and its value may be seen as the median

of the cluster. The inserted noise vertex act as a reverse engineering of the collapse, where each inserted noise would represent an element that belonged to the same collapse as the template vertex. Figure 45 provides an overview of our experiment.

Figure 45: Overview of our experiment plan for graph summarization

The noise insertion in the graph is made through insertions of new vertices (and edges connecting these vertices) with values between two existing vertices of the template graph. This type of insertion preserves the original graph layout since new vertices will always be between two existing vertices. Considering that each template vertex is used as the oracle for the summarization and represent a *collapse group*, then the added noise is required to have a compatible value to belong to the *collapse group*. Otherwise, the original premise that each template vertex represents a collapse of similar vertices would be invalid.

As mentioned before, we divided the experiment graphs to include two different numeric behaviors for the graphs. For the random noise values, the insertion of new noise vertex in the graph was accomplished by five steps: (1) define *n*, which is the number vertex to be added as noise; (2) for each noise vertex to be added, randomly pick a template vertex; (3) calculate the minimum distance to the closest template vertex neighbor from the selected template vertex and define this distance as three-sigma; (4) generate the random value for the noise vertex using a Gaussian distribution, where the median is the selected template vertex value and three-sigma is the minimum distance; and (5) create the noise vertex with the generated value and randomly insert it between the selected template vertex and one of its neighbors, inserting new edges accordingly.

We choose the Gaussian distribution because it allows generating values that are considered similar to the template vertex due to the probabilistic nature of the distribution. The minimal distance from the template vertex to another template vertex is the limiter in the Gaussian distribution (or the three-sigma), ensuring that 99.7% of the generated values will orbit around the median (template vertex value) with the maximum distance of three-sigma, according to the 3-sigma rule. Figure 46 illustrates the value distribution according to the 3-sigma rule. Moreover, there is a 6-sigma rule (HARRY; MOTOROLA UNIVERSITY PRESS, 1997) stating that 99.99966% of the values will be within the maximum distance of six-sigma. Furthermore, the minimal distance also ensures that the initial premise is valid, which has a sequence of similar values. If we use a value higher than the minimum distance, then its value

would be closer to another template vertex neighbor than the selected template vertex. This would either result in a splitting point or encapsulate the other template vertex inside the same *collapse group* if there are many more noise vertices with values higher than the minimum distance. We can impose this restriction because we are simulating an expansion of a collapsed group to try to reverse engineer the original members that led to that collapsed (template) vertex. Nonetheless, outliers are possible to be generated, since the 3-sigma rule only guarantees 99.7% of the generated values to be lower than the minimum distance. However, they will be few in numbers and sparse in the graph, which basically describes the meaning of an outlier.

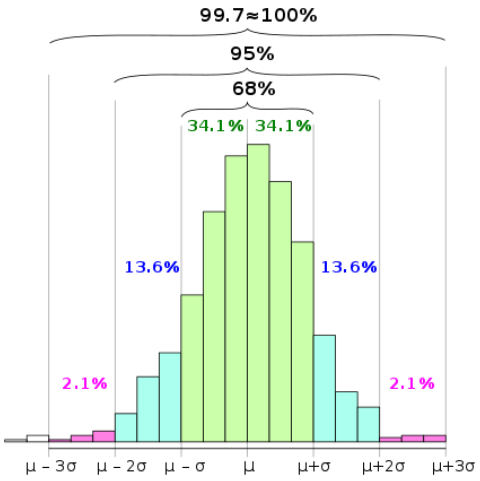


Figure 46: 3-sigma rule distribution of values. Source: Wikipedia[14].

The monotonic noise behavior is slightly different and was accomplished by only four steps: (1) randomly select a template vertex; (2) randomly select an edge that connects the selected template vertex; (3) generate a random value that is between the vertices' values from the edge; and (4) insert the noise vertex with the generated value between the vertices that belongs to the selected edge, replacing the edge with new edges to correctly connect the three vertices. The generated noise values will always follow the original behavior of the two original template vertices: if the second template vertex value is higher than the first, then all noise values between those two template vertices will follow an increasing function, otherwise, they will follow a decreasing function. The resulting graphs have smoother value transitions due to the monotonic nature of the noise value generation.

The last stage of each the experiment was the result analysis. We performed a statistical analysis over the results by means of hypothesis test in order to compare the obtained results of

[14] Source: https://en.wikipedia.org/wiki/68%E2%80%9395%E2%80%9399.7_rule

each collapse algorithm. An important factor for the design of the experiment concerns the definition of the significance level used during statistical analysis. We used a confidence interval (CI) of 95%, which translates to $\alpha = 0.05$, where α is the probability of rejecting the null hypothesis given that it is true (Type I error) (NORMAN; STREINER, 2012).

6.4 AUTOMATIC EXPERIMENT EVALUATION

In order to answer the proposed research question, we must analyze the reduction of the graph size after applying the automatic collapse of the graph and verify if the non-collapsed vertices represent events that have had great variations in the state. This analysis is done through the precision, recall, and F-measure metrics to evaluate the effectiveness of automatic collapse by comparing with the synthetic oracle that was used to generate the graphs.

6.4.1 MATERIALS AND METHOD

This experiment was executed through the use of synthetically generated provenance graphs as described in the previous section. We evaluated the three different categories of graphs to know which situations the proposed algorithms were more effective, considering that different domains can lead to graphs of different types. With this experiment, we can map which domains are most appropriate for each heuristic. Thus, we calculated the dependent variables (precision, recall, and F-measure) when applying the algorithm in each graph category (Direct Acyclic Graph, tree, and linear). Moreover, we also analyzed the results with two different numeric behaviors for the noise insertion (random and monotonic). This results in six different categories to be analyzed: (1) randomized DAG; (2) randomized tree; (3) randomized linear; (4) monotonic DAG; (5) monotonic tree; and (6) monotonic linear. In this experiment, we compare the collapse suggested by each algorithm with the template graph (*i.e.*, the synthetic oracle, which is an artificially created oracle), which represents the collapse baseline. We measured the dependent variables (precision, recall, F-measure) for each of the six different categories. The precision metric tells us how many of the generated *collapse groups* were correct in relation to the Oracle. The recall metric tells us how many of the correct *collapse groups* were generated by the algorithm. Finally, the F-measure tells us the overall performance of the algorithm based on the compromise of precision and recall metrics.

We defined a cluster to be correct based on the oracle if the proposed *collapse group* had only one oracle (i.e., template) vertex in it. For example, if we used a template graph with four template vertices to generate the noise graph and the algorithm collapsed all vertices from the noise graph into a single group, then there will be a 0% precision since it had zero correct

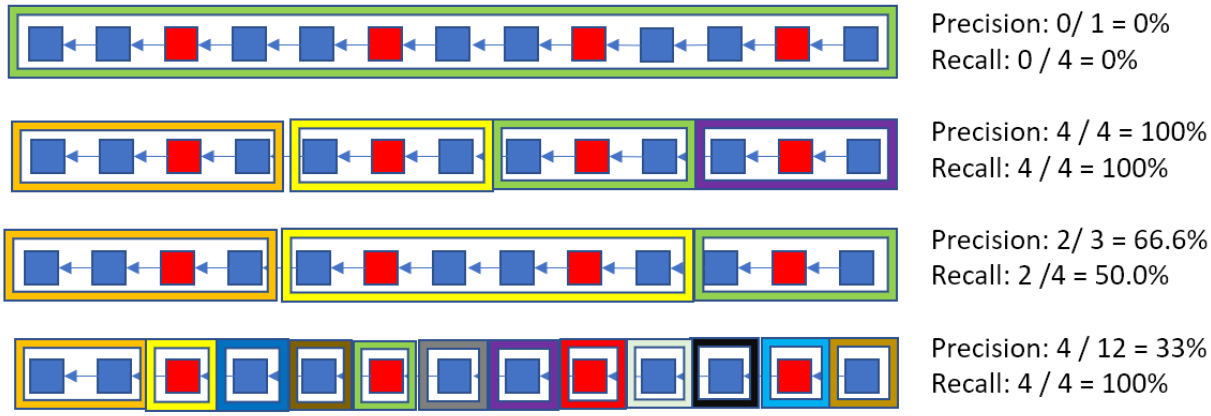


Figure 47: Examples of calculating the precision and recall for the experiment

collapse groups out of four, and its recall would also be 0%. Figure 47 illustrates some examples of the precision and recall computations for our experiment.

We first trained each algorithm through parameter tuning to find the best configuration using a smaller and separate training dataset. Then, we ran seven different iterations for each one of the six categories. Each iteration had forty template graphs that each spawned five noise graphs, totaling in 200 noise graphs to be analyzed for each iteration. The size of the noise graph grew by a factor of two with an initial size of ten times the number of vertices in the template graph (*i.e.*, 10x, 20x, 40x, 80x, 160x, 320x, 640x the size of the template graph). All four algorithms analyzed the same graph, in parallel, before generating the next graph of the experiment. This allows us to better measure their effectiveness since all algorithm result was based on the same graph used by the other three algorithms.

The experiment execution plan was divided into three stages: (1) Train the algorithms through parameter tuning in order to find the optimal configurable parameter values for each category, (2) execute the experiment using the resulting parameter values from stage 1, and (3) analyze the results. We trained all algorithms for each category in order to find the best values for their configurable parameters to maximize their F-measure result. Each training session used a smaller sample of randomized graphs. Nevertheless, the training session also used multiple template graphs to spawn different noise graphs during each iteration. The number of iterations was also reduced, removing some of the intermediate iterations for the training session.

During the second stage, we analyzed 200 different noise graphs in each one of the seven iterations, generating a total of 1,400 graphs for each category. Thus, for the randomized behavior, we analyzed the results from all four algorithms from 4,200 different graphs (*i.e.*, 1,400 DAG, 1,400 trees, 1,400 linear). Analogously, we also analyzed other 4,200 different

graphs for the partially monotonic behavior, using the same structure of seven iterations, where each iteration had forty different template graphs and five noise graphs spawned from each template graph. Table 2 describes the graph size used during each one of the seven iterations of the experiment for both randomized and partially monotonic behaviors. The results of each algorithm were compared with the synthetic oracle, which is the template graph used to generate the noise graph that was given to the algorithms, to calculate their precision, recall, and F-measure values. We then used F-measure to define the most qualified algorithm for each analyzed situation.

Table 2: Graph size for each iteration of the automatic experiment.

		Graph Size (vertices)					
		Random Noise			Monotonic Noise		
		DAG	Tree	Linear	DAG	Tree	Linear
Iteration	1	50	90	100	50	90	100
	2	100	180	200	100	180	200
	3	200	360	400	200	360	400
	4	400	720	800	400	720	800
	5	800	1,440	1,600	800	1,440	1,600
	6	1,600	2,880	3,200	1,600	2,880	3,200
	7	3,200	5,760	6,400	3,200	5,760	6,400

6.4.2 **RESULTS AND DISCUSSION**

First, we ran a normality test to verify if the data followed a normal distribution. According to the results of the Shapiro-Wilk test (SHAPIRO; WILK, 1965), the normality assumption was violated for all obtained results from the experiment since each dataset had a $p - value < 2.2 * 10^{-16}$. Therefore, non-parametric tests were adopted for statistical analysis. The non-parametric test used to compare the means was Wilcoxon Matched-Pairs Signed-Rank test (WILCOXON, 1945). Although there are other non-parametric tests, such as Chi-2 and Kruskal-Wallis, Wilcoxon Matched-Pairs Signed-Rank was chosen because it compares two means from two different samples against the same alternative hypothesis over the same (paired) observation (i.e., graph), which fits our experiment design.

Considering we have four different algorithms to compare and we decided to use Wilcoxon test, it is necessary to run six analyses: (1) DBSCAN vs IC; (2) DBSCAN vs VE; (3) DBSCAN vs ICVE; (4) IC vs VE; (5) IC vs ICVE; and (6) VE vs ICVE. We decided to use

the Bonferroni correction (DUNN, 1959) in the alpha-value to compensate, which translates to $\alpha = 0.00833$ since we have six comparisons. We adopted the following format for the hypothesis in our tests, naming *alg1* as the first algorithm used in the comparison and *alg2* the second algorithm used in the comparison:

$$H_0: \mu_{alg1} = \mu_{alg2}$$
$$H_1: \mu_{alg1} \neq \mu_{alg2}$$

It is possible to assert that there is a difference in mean if the null hypothesis is rejected. The null hypothesis is not rejected if the *p-value* is greater than the significance level α. In other words, there would not be enough evidence to assert a difference between results. When the null hypothesis is rejected (*p-value* < α), we can use the *box plots* to determine the superior method. All tests in this section were made using *R*, which is an open-source tool commonly used for statistical analysis.

The *box plots* of Figure 48 summarize the distributions of all four approaches for the 8,400 analyzed graphs. In these graphics, the boxes represent part of the central distribution, which contains 50% of data. Thus, the data scattering is proportional to the box's height. A black line inside the box represents the median. This way, 25% of the data is between the box's edges and the median. The median location indicates if the distributions are symmetrical in the experiments. Lastly, circles indicate outliers. The *box plots* for each algorithm measures the precision, recall, and F-measure of the algorithms.

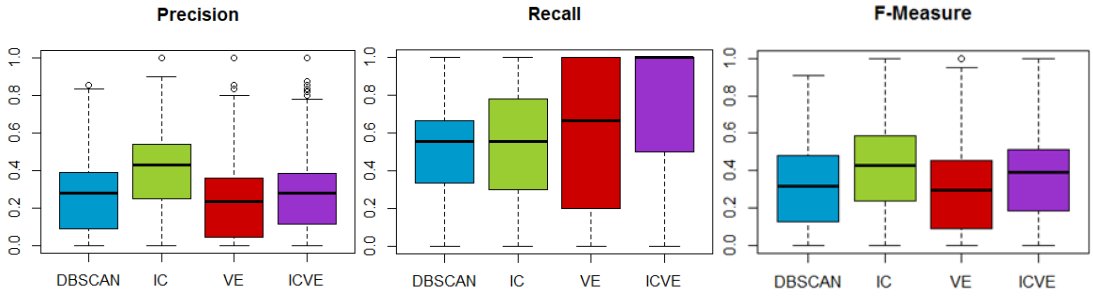


Figure 48: Box plot of the automatic results for each algorithm

The *box plots* show the VE algorithm having the lowest median (0.288) and IC with the highest median (0.414) for F-measure. However, all algorithms have high F-measure

Table 3: F-measure results from Wilcoxon test and Cliff's Delta effect size for the automatic experiment. The sign between parentheses represents if the CI is positive or negative when comparing the algorithms.

F-measure	DBSCAN vs IC	DBSCAN vs VE	DBSCAN vs ICVE	IC vs VE	IC vs ICVE	VE vs ICVE
p-value	2.2×10^{-16} (-)	1.291×10^{-16} (+)	2.2×10^{-16} (-)	2.2×10^{-16} (+)	2.2×10^{-16} (+)	2.2×10^{-16} (-)
Effect Size	-0.2587116 (small)	0.08436762 (negligible)	-0.1062623 (negligible)	0.3150409 (small)	0.1450793 (negligible)	-0.1806847 (small)

amplitudes. Looking at the precision *box plot*, we can see that IC has the highest median as well. However, IC has the lowest recall, while VE and ICVE have the highest recall of all algorithms, with ICVE winning due to his higher median at the maximum value. By analyzing the *p-values* from Table 3, the IC algorithm provided higher F-measure results than the other three algorithms (*p-value* < α), even after applying the Bonferroni correction. This is also supported by the effect size calculated using Cliff's Delta method, where IC has a small difference from DBSCAN and VE.

Considering the high amplitude of all algorithms, we decided to split the analysis into two groups: one considering only random noise graphs and another for using the monotonic noise graphs. Figure 49 illustrates the *box plots* of both groups for each algorithm. The *box plots* show that each algorithm had considerable different results for each group, especially the DBSCAN algorithm, showing worst results when using random noise and a much better result when dealing with monotonic noise. These observations can be confirmed by looking at Table 4, showing that DBSCAN indeed had worse results with random noise graphs and better results than VE and ICVE algorithms with monotonic noise. However, the ICVE algorithm proved to be better with random noise due to its higher recall and above average precision, while the IC algorithm proved to be better with monotonic noise since it got a higher precision in comparison with the rest. This is supported by the effect size between the algorithms shown in Table 4. The high amplitude of the algorithms, even after dividing into two groups (random and monotonic

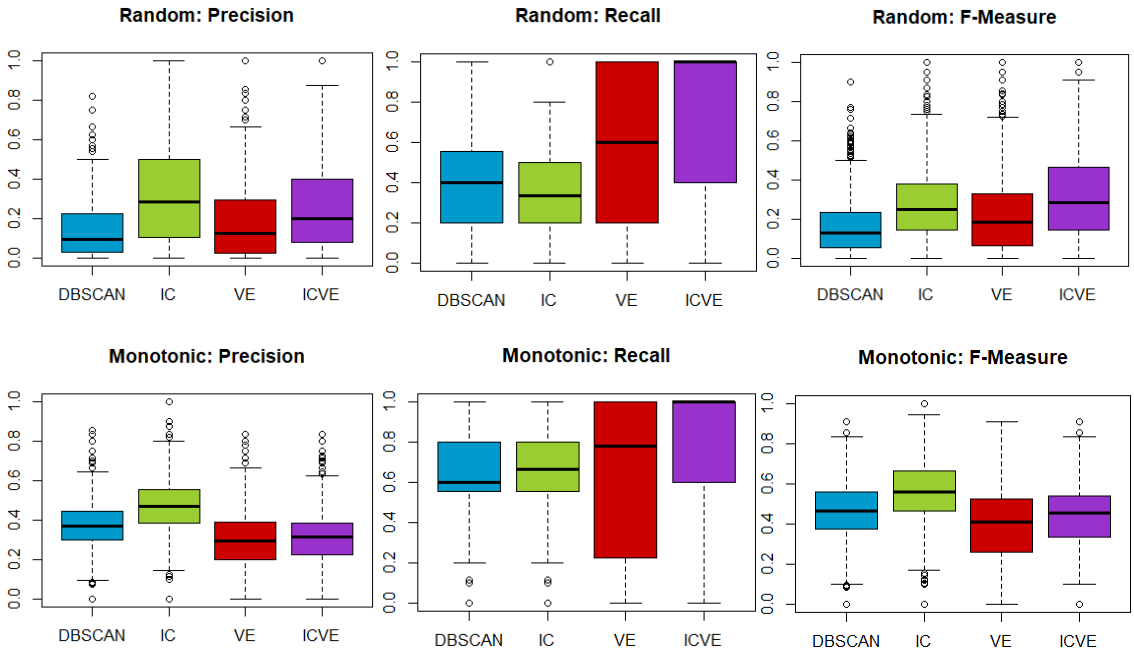


Figure 49: Divided *box plots* of the automatic experiment for the random and monotonic noise.

Table 4: F-measure results from Wilcoxon test and Cliff's Delta effect size for the automatic experiment divided into two groups: Random and Monotonic noise

F-measure		DBSCAN vs IC	DBSCAN vs VE	DBSCAN vs ICVE	IC vs VE	IC vs ICVE	VE vs ICVE
Random Noise	p-value	2.2×10^{-16} (-)	2.2×10^{-16} (-)	2.2×10^{-16} (-)	2.2×10^{-16} (+)	2.2×10^{-16} (-)	2.2×10^{-16} (-)
	Effect Size	-0.3997387 (medium)	-0.1289706 (negligible)	-0.4000328 (medium)	0.2138566 (small)	-0.08052234 (negligible)	-0.2605855 (small)
Monotonic Noise	p-value	2.2×10^{-16} (-)	2.2×10^{-16} (+)	2.2×10^{-16} (+)	2.2×10^{-16} (+)	2.2×10^{-16} (+)	2.2×10^{-16} (-)
	Effect Size	-0.3449252 (medium)	0.2465062 (small)	0.1366312 (negligible)	0.5193248 (large)	0.4567062 (medium)	-0.1076311 (negligible)

noise) is due to the seven iterations used in the experiment, which increases the graph sizes by doubling the graph's size from the previous iteration.

These results indicate that the IC and ICVE algorithms provide overall better collapses than the other algorithms, including the DBSCAN, in both random and monotonic noise graphs, independently of the graph size. Thus, we can conclude that the inter-cluster verification does indeed provide better clusters when considering DAG, Linear, or Tree graphs due to its nature – it better adapts to the neighborhood by shaping the cluster through distance checks to the farthest existing member already inside the cluster instead of checking only the direct nearest member of the candidate neighbor. This results in a behavior that creates close-knit clusters and consequently avoids overgrowing the cluster to encompass the majority of the graph when nodes have small and progressive variations in relation to neighbors.

Meanwhile, the DBSCAN compares the candidate neighbor only with the nearest member of a cluster and therefore tries to compensate this deficiency by reducing the distance threshold in order to create correct clusters according to the oracle. This results in an increased recall at the cost of precision since it will create many clusters and only a few of these are equivalent to the ones in the oracle. On the other hand, the ICVE stayed behind the IC algorithm because it refined too much in the cluster construction, resulting in more close-knit clusters than necessary and, consequentially, farther from the oracle than IC. This is even more apparent in the monotonic noise graphs by looking at the effect size metric, where the IC algorithm provided the best results in comparison to all the other algorithms, with a good margin. However, the variable epsilon approach (VE), when used alone, proved to be detrimental and achieved only better results than the DBSCAN when dealing with monotonic noise, which would correlate to a graph with smooth value transitions.

Figure 50 and Table 5 demonstrate the results when we break down the results for each graph type instead of noise types. These results indicate that the IC and ICVE algorithms provide overall better collapses than the other algorithms for all three graph types,

independently of the graph size. However, despite these results, the difference between DBSCAN and ICVE algorithms is negligible.

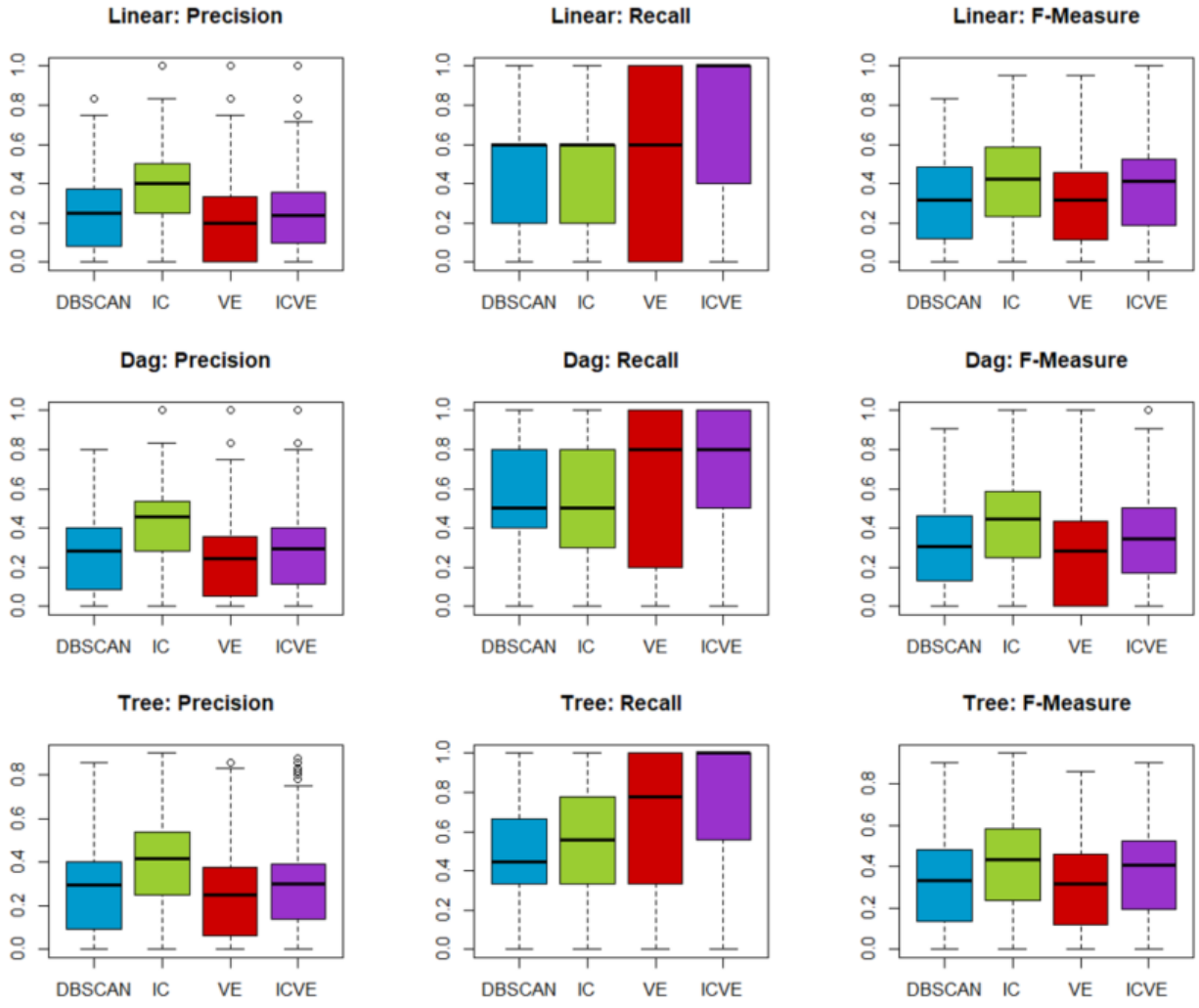


Figure 50: Divided *box plots* of the automatic experiment for each graph type

Table 5: F-measure results from Wilcoxon test and Cliff's Delta effect size for graph types

F-measure		DBSCAN vs IC	DBSCAN vs VE	DBSCAN vs ICVE	IC vs VE	IC vs ICVE	VE vs ICVE
Linear	p-value	2.2×10^{-16} (-)	6.101×10^{-2} (+)	2.2×10^{-16} (-)	2.2×10^{-16} (+)	1.313×10^{-14} (+)	2.2×10^{-16} (-)
	Effect Size	-0.2539269 (small)	0.07150306 (negligible)	-0.1352889 (negligible)	0.3000265 (small)	0.1064872 (negligible)	-0.2057124 (small)
DAG	p-value	2.2×10^{-16} (-)	1.324×10^{-7} (+)	1.557×10^{-7} (-)	2.2×10^{-16} (+)	2.2×10^{-16} (+)	2.2×10^{-16} (-)
	Effect Size	-0.2711593 (small)	0.1190957 (negligible)	-0.05111339 (negligible)	0.3476515 (medium)	0.2082344 (small)	-0.156294 (small)
Tree	p-value	2.2×10^{-16} (-)	3.53×10^{-2} (+)	2.2×10^{-16} (-)	2.2×10^{-16} (+)	2.2×10^{-16} (+)	2.2×10^{-16} (-)
	Effect Size	-0.2553388 (small)	0.06183724 (negligible)	-0.1349087 (negligible)	0.296004 (small)	0.1208685 (negligible)	-0.1890452 (small)

6.4.3 **THREATS TO VALIDITY**

We identified internal and external factors that may influence the results. In relation to internal validity, the graph generation algorithm and noise insertion can affect the results because they are all synthetic in nature. Another threat is related to the automatic evaluation of the generated clusters from each algorithm with the template graph. We are not aware of other work that proposed such method of automatic analysis, which synthetically expand graphs with noise insertion and then use clustering algorithms in order to omit the noise vertex, trying to return to the original graph, using precision, recall, and F-measure to evaluate each algorithm in order to determine the best solution.

Lastly, another threat is related to the training sessions of the algorithms. All algorithms were trained for each category with varying graph sizes, selecting the best overall parameters values for it depending on the graph type and noise insertion (i.e., random or monotonic). We did not split the training sessions to find the best configurations for each different graph sizes used by the iterations of each category. This can change the results of the algorithms when the graph size differs too much between iterations, especially when comparing the initial iterations with the two last iterations, where the graphs have thousands of more vertices than the previous iterations. Thus, the training session selected the parameters that provided the best overall quality for clustering the graph independently of its size, which could have clusters differing from a few dozen vertices in small graphs to hundreds of vertices in larger graphs since only the noise varied in each iteration, while the number of correct clusters in the Oracle maintained the same independently of the graph size. This behavior is reflected in the high amplitude in all algorithms and resulted in having similar precision and recall in all algorithms, even after splitting the experiment results between the two noise groups (random and monotonic). However, it might be possible to achieve better tuning for each algorithm by changing some of the factors used during the training session. For example, we aimed at maximizing the overall F-Measure for each algorithm and graph type, independently of the graph size. Furthermore, since training is an exhaustive and time-consuming operation, we also limited the training sample size to only 5 iterations, with 20 oracle graphs in each iteration and each oracle graph generating only three noise graphs to be used for training. It might be possible to achieve better tuning by increasing the training sample size and specializing each training session to a specific graph size category.

Regarding external validity, we mitigated sample bias by randomly generating multiple different template graphs for each iteration of the experiment. Each template graph was also

used to spawn multiple noise graphs, where each one was used by all four clustering algorithms, mitigating any bias in the algorithm evaluation since all four algorithms were using the same noise graphs during each step of the experiment.

6.5 EXPERTS EXPERIMENT EVALUATION

Our research question aims at identifying the most effective similarity summarization algorithm for reducing the information to be analyzed, while still preserving the graph semantics. To do so, we also executed qualitative experiments with human judges to evaluate each one of the four algorithms in order to select the most appropriate, based on their opinion, for each category of the graph.

6.5.1 MATERIALS AND METHOD

In this experiment, due to the nature of using human beings as an oracle, we had to use a smaller set of graphs with hundreds of vertices instead of thousands of graphs with thousands of vertices like in the previous experiment. We used the same types of graphs from the automatic experiment: three graph types (DAG, tree, linear) and two graph characteristics (random and monotonic noise). The combination results in six different categories to be analyzed by each subject: (1) randomized DAG; (2) randomized tree; (3) randomized linear; (4) monotonic DAG; (5) monotonic tree; and (6) monotonic linear. Furthermore, we used the same graph generation technique described in the previous section to generate all synthetic graphs, with only one numeric attribute for the vertex value. Thus, the similarity between neighbor vertices would be based only on this numeric value.

We decided to use two different samples for each one of the six categories to reduce bias, resulting in having the judge to look and analyze 48 graphs in total, since each sample consists of the original graph and other four collapsed graphs, one for each algorithm. However, at the same time, we could not increase this number to three samples since each new sample would result in an additional 24 graphs to be analyzed by the judge. Then, the subject selects one of the four collapsed graphs that represents, in his opinion, a good collapse for that specific case.

In order to reduce the length of the experiment, we previously trained all algorithms using the same principle from the automatic experiment since we were also using synthetic graphs and also considering the graph sizes that were going to be used in this experiment. Thus, the judge does not need to fine-tune each algorithm for each graph and only need to select the

algorithm that generated the best result according to his opinion. We also used gradient colors instead of numbers to represent the vertex's value in the graph, in order to make easier for the subjects, resembling what a tool such as *Prov Viewer* would show. Red gradient represents negative values, green gradient represents positive values, and white gradient represent values close to zero.

We presented all the provenance graphs in a comprised way, with the original graph at the top left. Figure 51 illustrates the format used in the experiment, where the top left graph (mostly using only green, red, and white gradients) is the original graph and the other four colored graphs are the summarization results from each algorithm. Note that each output section of the sheet has two graphs: a graph with colored borders, showing the composition of each *collapse group*, and a summarized graph, which is the result after following the proposed collapses. Sequential vertices that had the same border color belong to the same *collapse group*. For example, in the output "C" from Figure 51, we can see a chain of blue-colored vertices near the middle of the graph. All these blue-colored vertices belong to the same *collapse group* since they all share the same border color. Thus, each colored chain of vertices represents a different *collapse group*. Vertices with colored borders that appear in the smaller summarized graph

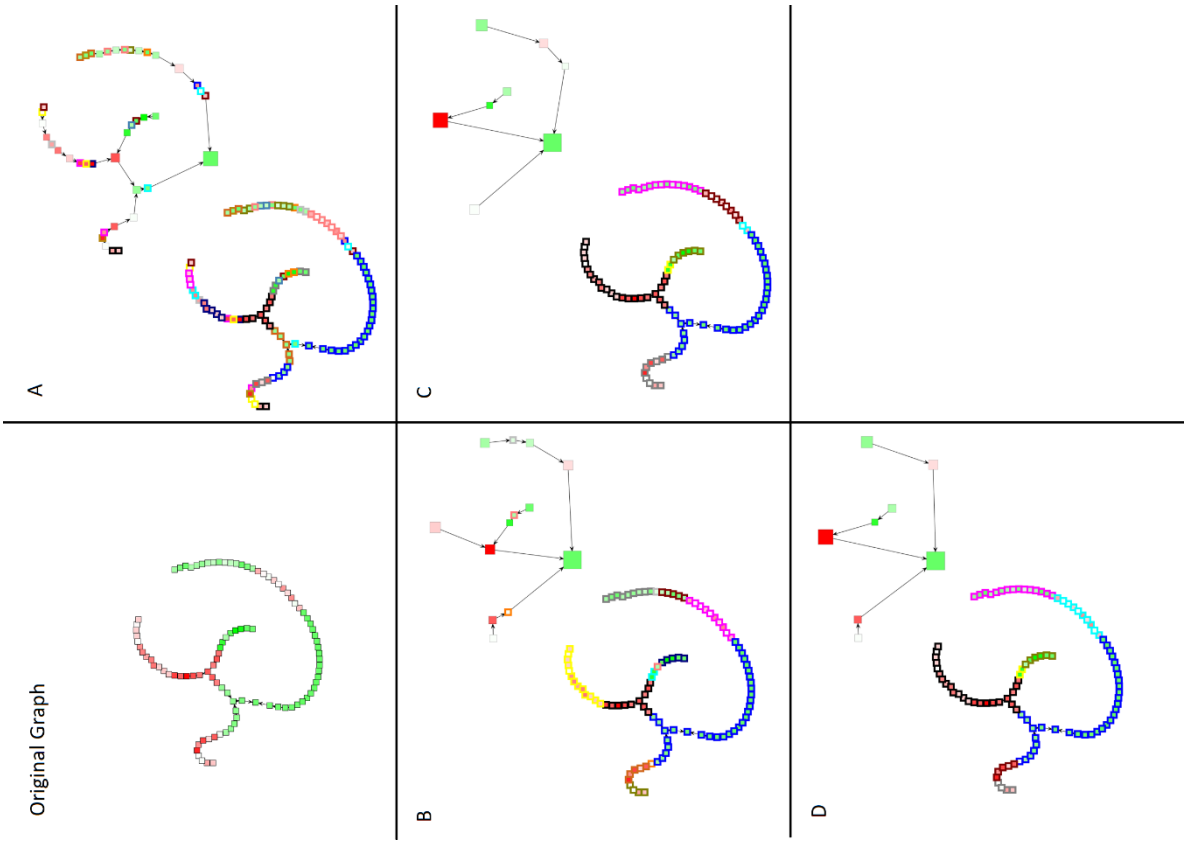


Figure 51: Example of a sheet given to the judge for his analysis of the algorithms.

represent outliers, which are *collapse groups* comprised of only a single vertex. Moreover, the size of the vertex in the summarized graph is proportional to the number of vertices in the *collapse group* that generated it. Thus, we can see in the same example that the blue chain of vertices from letter “C” generated a big green vertex in the summarized graph due to the high quantity of vertices in the chain and their predominant color in the original graph is a bright green. We decided to use these printed versions of the graph to expedite the analysis process, allowing the judge to compare side-by-side the different outputs.

For this experiment, we decided to use a simple vote process: each subject should point which category represents the most appropriate algorithm in his opinion. Thus, the experiment execution was divided into two stages: (1) a pilot experiment to detect any issues that needed to be addressed and (2) the experiment itself. During the pilot, volunteers were required to analyze each graph and select one of the four algorithms, picking the one that proposed the most appropriate *collapse group*s in his/her opinion. The pilot experiment was applied to three volunteers from the university.

In order to avoid biased answers, we randomized the algorithms order, forcing the subject to analyze all the four algorithms' output. We also adopted a speak-aloud approach in order to identify how each volunteer reached their final decision. Their recorded decision-making process can also be used to fine-tune the algorithms in the future. We incorporated this approach because the volunteers were not giving any feedback to back up their selection during the pilot.

Considering that all twelve graphs were synthetic, we decided to add two new graphs in the experiment. These graphs were real provenance graphs generated from a gameplay session of a racing game using the PinG approach. Both provenance graphs belonged to the same game session, and thus had the same vertices. They differentiated only in the displayed graph layout. The first graph used a simple graph layout (the same used for all the previous graphs) where we omit all the domain information, displaying it as another synthetic domain-less graph, with each vertex having a single numeric value without any semantics. The second graph used a spatio-reference layout, placing all vertices in their spatial location when the action represented by the vertex was executed. Moreover, we explained to the volunteer the semantics of this graph, as well as the domain it belongs. By asking the volunteer to analyze the same provenance graph twice, but having a different perception from each, we were able to discern if the graph domain and the spatio-referencing of the data generated a significant impact in the summarization process.

Lastly, we made a final change in the experiment by introducing a post-experiment questionnaire[15], designed to be answered at the end of the session and allowing us to gather additional feedback and further insights from the subject that were not captured by the talk-aloud strategy. This questionnaire included two questions related to the final two real provenance graphs, allowing us to verify if the domain information had any impact on the analysis: (1) *The positioning of the vertices according to the spatial information from where they were executed helped in its decision process?* (2) *After knowing the meaning of the colors and the context of the data, did it influence the decision process?*

After changing the original experiment structure used during the pilot, the resulting experiment plan was divided into three stages: (1) Generating the graphs, (2) running the experiment with judges, and (3) analyzing the results. We executed the first stage before running the experiment. In this stage, we trained all the algorithms for each category. Then, we generated two graphs for each category and ran the algorithms for each graph, creating their corresponding paper sheets with the original graph and the four different outputs.

The next stage was the experiment execution with the judges. We applied the experiment with fifteen volunteers closely related to computer science. Figure 52 illustrates the characterization of the judges by their degree. It is important to note that three of the Ph.D. volunteers were considered specialists in graphs. As we can see by looking at this figure, 47% of the volunteers had a Ph.D. degree and were professors in the university. The remaining volunteers were divided in 33% having the Master's Degree and currently working on their Ph.D. degree and the remaining 20% being students under the master's program. Table 6 shows in more detail the characterization of each volunteer.

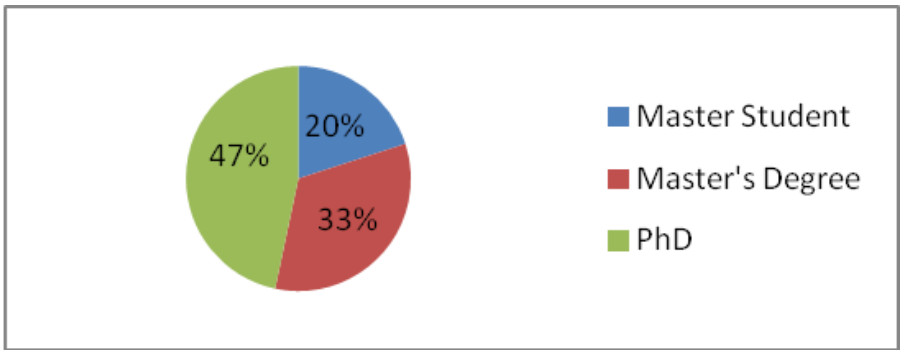


Figure 52: Volunteers' characterization chart

[15] All questionnaires are available at: https://github.com/gems-uff/prov-viewer/tree/master/Documents

Table 6: Volunteers' characterization table

Subject	Academic Education	Conclusion Year	Graph Knowledge
P1	Master's Program	2017	No formal knowledge
P2	PhD	2010	Studied
P3	Master's Degree	2017	No formal knowledge
P4	PhD	2012	Studied
P5	Master's Degree	2019	Studied
P6	PhD	2011	Studied
P7	Master's Degree	2016	Studied
P8	PhD	1998	Expert
P9	Master's Program	2017	Studied
P10	PhD	2014	Expert
P11	Master's Degree	2017	Read About
P12	PhD	2003	Read About
P13	PhD	1998	Expert
P14	Master's Degree	2017	Studied
P15	Master's Program	2017	Read About

Each experiment had an average duration of one hour and a half and was conducted individually due to the talk-aloud strategy. The volunteers analyzed 14 paper sheets, where each contained one of the fourteen graphs and the resulting outputs from each algorithm (as illustrated in Figure 51), marking their answers in a spreadsheet. The post-experiment questionnaire was handed to the volunteers after they finished analyzing all the 14 sheets.

6.5.2 **RESULTS AND DISCUSSION**

Figure 53 summarizes the results of the experiment for all four algorithms considering all twelve synthetic graphs analyzed by the fifteen judges. By analyzing the box plot, we can visually see that the IC algorithm had better results than any of the other three algorithms, confirming the results of the automatic experiment. From the box plot, we can see that the graph

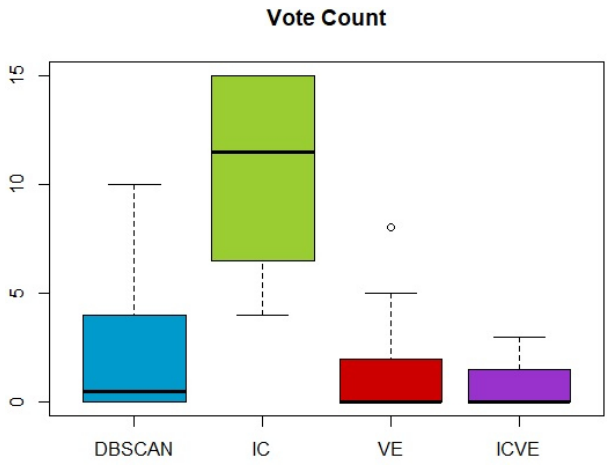


Figure 53: Box plot of the judge's results for each algorithm

in which IC received fewer votes had votes from five volunteers. On the other hand, there is at least one graph that IC received all the votes from volunteers. The DBSCAN algorithm was ranked in the second position with some situations almost reaching the median of IC. However, it is only possible to assert this assumption by running statistical tests.

Similar to the automatic experiment, we ran a normality test using Shapiro-Wilk test (SHAPIRO; WILK, 1965) and noticed that the collected data does not follow a normal distribution. Therefore, once again we used the Wilcoxon Matched-Pairs Signed-Rank test because it compares two means from two different samples over the same observation (*e.g.*, graph), which fits our experiment design.

Again, we ran six analyses to compare the algorithms: (1) DBSCAN vs IC; (2) DBSCAN vs VE; (3) DBSCAN vs ICVE; (4) IC vs VE; (5) IC vs ICVE; and (6) VE vs ICVE. Considering that we are using six comparisons, we also decided to use the Bonferroni correction in this experiment, which translates to $\alpha = 0.00833$. We adopted the same format from the automatic experiment for the hypothesis in our tests, naming *alg1* as the first algorithm used in the comparison and *alg2* the second algorithm used in the comparison:

$$H_0: \mu_{alg1} = \mu_{alg2}$$
$$H_1: \mu_{alg1} \neq \mu_{alg2}$$

By analyzing the *p-values*, CI, and the effect size from Table 7, we can see that the IC algorithm provided better results, even when applying the Bonferroni correction. Therefore, the null hypothesis was rejected in all comparisons involving the IC algorithm. However, there is not enough evidence (*p-value* > α) to assert the difference between results when comparing the other algorithms to provide a ranking, only enough to know which of all four is the best. This finding matches with our initial visual analysis of the box plot from this experiment (Figure 53), where we could clearly see that algorithm IC had better results than all other analyzed algorithms. Furthermore, this result also coincides with the initial findings from the automatic experiment, where the IC algorithm was also the one that had a highest F-measure. Thus, the initial findings from both experiments are in tune.

Table 7: Results from Wilcoxon test for the judge experiment. The sign between brackets represents if the CI is positive or negative when comparing the algorithms.

F-measure	DBSCAN vs IC	DBSCAN vs VE	DBSCAN vs ICVE	IC vs VE	IC vs ICVE	VE vs ICVE
p-value	**0.008062 (-)**	0.6741(-)	0.1434 (-)	**0.004673 (+)**	**0.002328 (+)**	0.4568 (+)
Effect Size	-0.8611111 (large)	0.1666667 (small)	0.2291667 (small)	0.9236111 (large)	1 **(large)**	0.05555556 (negligible)

However, if we examine the box plot closely, we can see that the DBSCAN amplitude almost reaches the IC median. This might mean that in some cases the DBSCAN can be at least equal to IC. If we break down to the results for each category, as illustrated by Figure 54, we can see a large difference between random and monotonic noise results. We did not use box plots this time because there are only two graph samples for each category, which would make statistical analysis unfeasible. The IC algorithm dominates in monotonic noise graphs and is adequate in random graphs, thus making it an ideal general-purpose algorithm for summarizing provenance graphs. Meanwhile, the ICVE algorithm appears to be always inappropriate. The judges did not like this algorithm because in most cases, as the majority of the judges summarized, it is "*too sensitive to changes and creates too many different clusters, not summarizing much the data*" (P8).

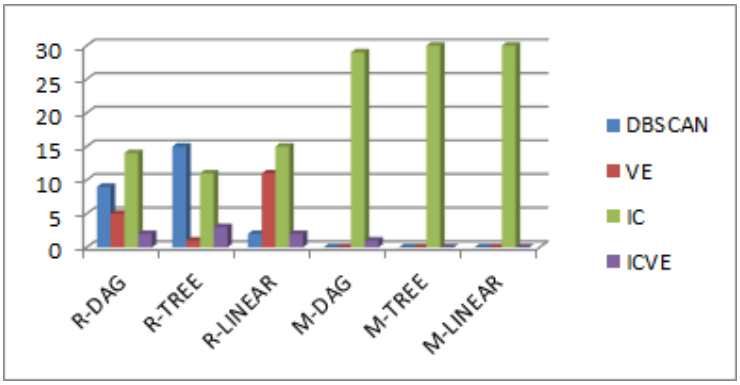


Figure 54: Experiment results from each one of the categories in the experiment

Considering the random categories, the DBSCAN appears to be only inappropriate when dealing with linear graphs. This occurred because it overextended the reach of a cluster, grouping almost half of the graph in a single cluster because it failed to detect the slope variations (both increasing and decreasing), stopping only when encountered an extremely sharp slope. As one of the experts said (P13): "*by identifying this entire region, I remove it from the dispute*" when he/she was referring to half the graph that the algorithm failed to divide that was composed of a light green segment followed by a dark green, then another light green segment, and ending with a whitish red segment. Despite VE showing to be inappropriate with tree-graphs, the resulting clusters were very similar to those from DBSCAN, showing small differences that led them to not be chosen when being in doubt between DBSCAN and VE.

Table 8 illustrates the statistical analysis of the Random Noise graphs used in this experiment. We ommited the analysis of the Monotonic Noise since the IC algorithm dominated in this situation. The results show that the IC algorithm is indeed better than VE and ICVE algorithms. However, there is not enough statistical evidence to make claims with the DBSCAN

since the null hypothesis was not rejected (p-value = 0.0855 > 0.00833). Overall, this finding also matches with the ones from the previous experiment, where IC algorithm proved to be better in both monotonic and random noise.

Table 8: F-measure results from Wilcoxon test and Cliff's Delta effect size for the random noise of the expert experiment. Used Bonferroni Correction (0.00833).

F-measure		DBSCAN vs IC	DBSCAN vs VE	DBSCAN vs ICVE	IC vs VE	IC vs ICVE	VE vs ICVE
Random Noise	p-value	8.55×10^{-2} (-)	1.721×10^{-2} (+)	9.764×10^{-4} (+)	2.351×10^{-3} (+)	1.529×10^{-6} (+)	4.288×10^{-2} (+)
	Effect Size	-0.1555556 (small)	0.1 (negligible)	0.2111111 (small)	0.2555556 (small)	0.3666667 (medium)	0.1111111 (negligible)

Figure 55 illustrates the results only from the experts' point of view and Figure 56 provides a summary of these results in the form of box plots. None of the expert judges selected the ICVE algorithm in any of the categories, corroborating that it is the least favorable algorithm among the judges. This figure also shows that IC is generally the most appropriate algorithm, with the exception when dealing with the random tree category, where the DBSCAN showed better results. Nonetheless, these results are similar to those presented in Figure 54, where the random tree was the only case when IC lost to another algorithm, which was also the DBSCAN. However, despite these observations for the random graphs, there are not enough statistical data to determine the most appropriate algorithm for each type of graph since each category only had two graph samples and 15 human judges, resulting in only 30 samples and the ideal is to have at least 50 samples.

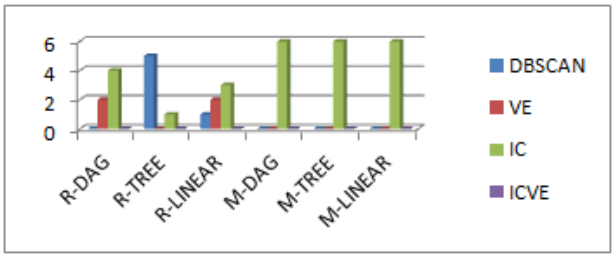


Figure 55: Expert's results

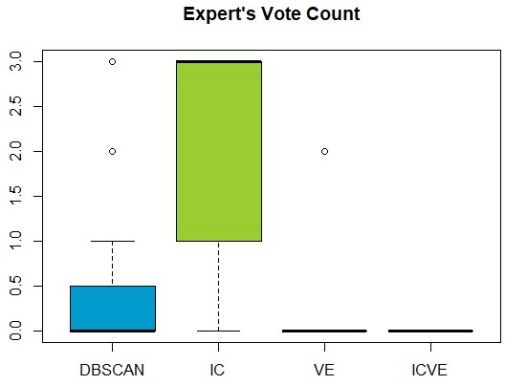


Figure 56: Box plot of only the three experts for each algorithm

As mentioned before, all those 12 graphs were synthetic in nature, although based on structures frequently found on provenance generated from games. However, in the same experiment, we had one real provenance graph with two different visualizations, which was based on a car game. We added this graph to have an initial analysis of how the algorithms would behave in a real scenario. The first time it appeared in the experiment, the judges treated it like any of the other synthetic graphs since the real game data looked exactly like any other of the artificially generated graphs from the templates and no additional information was given to them regarding the origin of the graph. However, unlike the previous graphs, the car's provenance graph only contained green colored vertices due to the nature of the game since the value represented the car's speed. Furthermore, since only the player was present, the resulting provenance graph was from the linear type. Figure 57 illustrates the results before and after revealing the details about the graph, where "Linear-Car" represents the judges' first interaction with the graph, thinking it was a synthetic graph, and "Geo-Car" represents their second interaction, where they had domain information and the graph's vertices were displayed over the race track map, showing their exact locations.

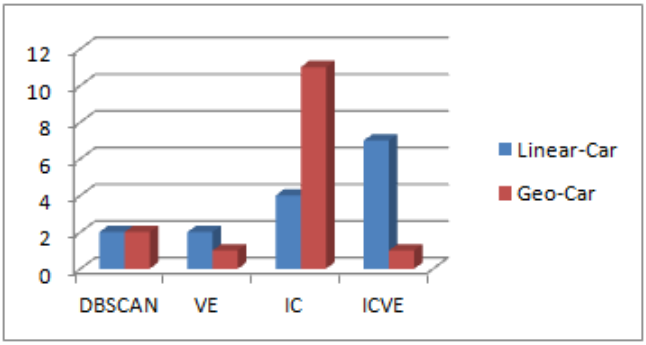


Figure 57: Results from the real provenance graph

Surprisingly, almost half of the judges picked the ICVE algorithm on the "Linear-Car" graph. However, all of them reported difficulties when analyzing the graph due to the higher number of vertices and also because all vertices were green-colored. After knowing the graph origins and what it represented, thirteen of the judges selected another algorithm when analyzing the "Geo-Car" graph, with the majority choosing the IC algorithm. As the majority of the judges said, knowing additional data information, such as domain, changed their perception because "*the contexts made me* (judge P2) *think about the causes of variation*". Moreover, the judges also agreed that, in this specific case, geo-referencing the data helped them in their decision-making process because "*the relative position of the vertices in relation to their neighbors helps"* (P8) when they were trying to see the speed difference between vertices due to their distance. Moreover, some judges were also in agreement that "*the context*

of the game shows that position matters to analyze the number of curves (from the race track), *which impacts in the performance analysis* (of the player)" (P1). However, one of the experts (P10) said that he "*abstracted the application"* and *"the additional information didn't impact in my (his) analysis*". Nevertheless, that same specialist selected the IC algorithm in both graphs (Linear-Car and Geo-Car). Figure 58 shows the "linear-car" graph superimposed on the race track, which was shown to the judges when explaining the graph's domain, and Figure 59 show both graphs presented to the judges for their evaluation. Notice that we removed the background from Geo-Car after presenting it to the judges, so they could easily detect and contrast the vertices to select one of the four collapses. However, they could consult the original Geo-Car graph (with the race track background) anytime.

Figure 58: Provenance Graph from the racing game used in the experiment. Figure (a) illustrates the "linear-car" graph superimposed in the race track, generating the "Geo-Car" visualization and (b) illustrates an example of one of the outputs from the IC algorithms. Vertices color represent the car's speed.

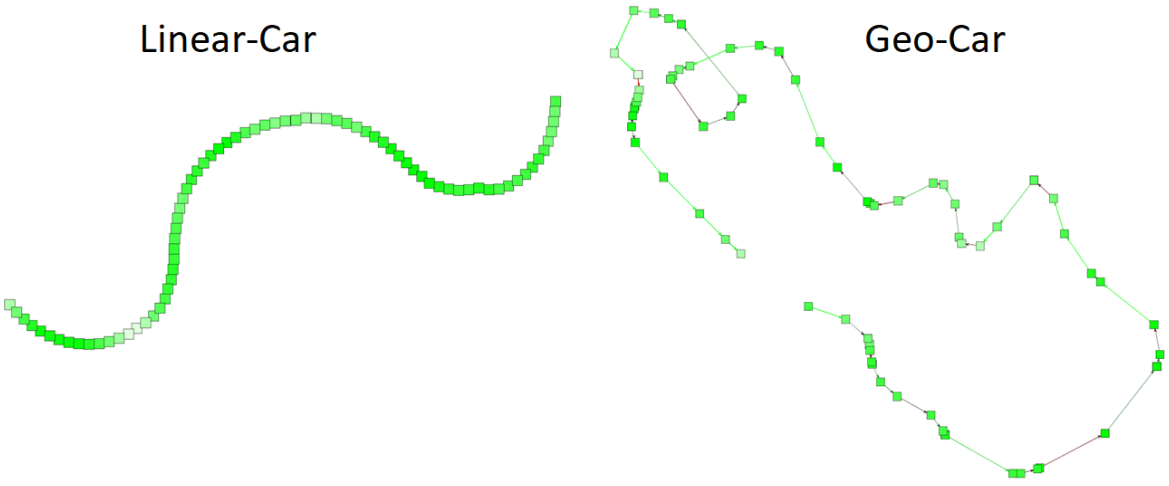


Figure 59: Linear-Car and Geo-Car graphs presented to the judges. Both graphs are the same as the only difference being in the vertex positioning.

These results point out that the IC algorithm provides overall better collapses than the other algorithms in both random and, in particular, monotonic noise graphs. However, when considering only Random Noise graphs, the DBSCAN tied with the IC algorithm and there were not enough data to determine which is the best solution. These results match with the ones from the automatic experiment, where the IC algorithm also proved to be the better algorithm. Therefore, with the findings from both the experiments, we can assume that the IC algorithm is more adequate than the other algorithms for clustering DAG, Tree, and Linear graphs. Furthermore, since the results from the automatic experiment matched with the one using human judges, we can also conclude that this automatic validation is also a viable method for measuring clustering effectiveness of the algorithms.

6.5.3 **THREATS TO VALIDITY**

Despite the care in reducing the threats to the validity of the experiment, there are factors that can influence the results. In relation to internal validity, the selection of participants can affect the results because of the natural variation in human preference in picking the answers, since there are no right or wrong answers in this experiment. Furthermore, the experiment was executed with volunteers since they generally are more motivated for executing tasks. Any volunteer could choose to be dismissed from the experiment and be released earlier. One possible threat is related to each individual perception of different colors and their shades in the graph. To minimize this problem, we also printed different versions of the material for color-blind individuals. Nonetheless, one of the experts in Visual Computing correctly remarked that the different shades used (i.e., red, green, and white) could also be a threat due to their different chromatic distance when analyzing the graphs.

Another threat is related to graph sizes used. Unlike the automatic experiment, we used graphs that could be considered small, with around a hundred vertices, in order to not confuse the volunteer during the analysis.

6.6 RELATED WORK

Our related works were selected from those used by the game industry for clustering and summarizing gameplay telemetry data and other graph-based research for visualization of gameplay data that uses some kind of clustering technique to group graph nodes. We are only interested in this moment in using summarization for noise reduction and aiding the visual exploration of the graph.

Play-Graph (WALLNER, 2013), and its follow-up system PLATO (WALLNER; KRIGLSTEIN, 2014), is a graph-based approach to formally describe and visualize game session data by using a graph visualization. It has multiple variables and their interrelations along with the temporal progression of players. It uses spatial information to render the graph in a game scene and cluster nearby nodes to form a single node that provides statistical information in the scene's region (*e.g.*, player race distribution at that location). However, its clustering method disregards temporal information and indiscriminately cluster similar states that are spatially close, losing the real sequence of events.

Another approach is *Playtracer* (LIU et al., 2011), which is a visual tool designed to illustrate how groups of players move through the game space. Thus, *Playtracer* aids the designer by showing common pathways and alternatives that players used to succeed or fail in their tasks, identifying pitfalls and anomalies in the scene. However, *Playtracer* does not take into consideration temporal information or the actual game map. The temporal information would allow stating the order of events in the game, shedding more light in the player's behavior, while the map of the scene would show exactly where these pathways, pitfalls, and game anomalies were in the game. Moreover, in order to solve problems related to the number of visible states, *Playtracer* uses an aggressive technique to cluster nearby states together to make a cleaner visualization, disregarding sequence of events and thus forming cycles in the graph.

Both Play-graph and Playtracer approaches are focused on state-changes analysis and therefore deal with a different type of game data, which is much coarser grained than event-based approaches. Furthermore, due to the different type of telemetry tracking (i.e., states instead of events), the tracked data possess much less variance since the tracking is limited to visited states instead of tracking the sequence of executed actions. Nevertheless, those approaches use Quality Threshold clustering technique, which is very similar to the DBSCAN clustering algorithm that we used to compare our algorithms.

Other related works include common approaches adopted by the game industry and game research, such as heat maps (DRACHEN; CANOSSA, 2009b) or trajectory analysis (BAUCKHAGE et al., 2014; MILLER; CROWCROFT, 2009; PAO; CHEN; CHANG, 2010), which display paths in a map. These approaches use visualizations based on the evaluation of one or two variables from the game data, providing an easy to read and intuitive interpretation of the data distribution. However, heat maps only aggregate variables that follow specified restrictions (*i.e.*, death locations) to show density distribution over the scene. Similarly,

trajectory maps also aggregate equivalent paths. Nevertheless, they do not provide any insight on the executed actions, hindering influences between events. However, we did not compare them with our proposed algorithms because they are used for a different type of data analysis. The heat map is commonly used to aggregate data within the same 2D spatial region to analyze the most visited sections of the map by players or can also be used for specific occurrences such as places where players died the most. Similarly, trajectory analysis is used to map the player's navigation in the game scenario.

Clustering behavioral data is a recent interest in the field (BAUCKHAGE; DRACHEN; SIFA, 2015) due to game analytics still being a relative new idea. Therefore, there are almost no study or new techniques for clustering behavioral data. Furthermore, most of the studies are focused on player profiling. There is a recent study (SAAS; GUITART; PERIÁÑEZ, 2016) that uses the Ward's minimum variance method (JR, 1963). However, the clustering is used to classify player profiles based on a few variables such as the amount of time spent in the game, number of sessions, total number of actions performed, and the number of in-game app purchases.

6.7 FINAL CONSIDERATIONS

This chapter proposed new graph algorithms based on the DBSCAN that take into consideration the temporal sequence of information of a provenance graph to summarize tracked data to a more manageable size through the usage of collapses strategies. None of the existing approaches considered also using temporal information in their collapse strategies. These collapses intend to reduce the overall graph size by hiding sections in the graph that alone were not significant enough to produce a meaningful impact on the game and did not offer any useful information for analysis.

Our proposal enhances the identification of sections or vertices that are different from its neighbors or the expected behavior. In a game context, vertices that were not collapsed represent drastic changes in the game state and are worth for displaying at the analysis, while all collapsed ones fluctuate around the same state for a given attribute. Therefore, the resulting collapsed graph is useful to confirm the initial hypothesis of sections that the player had difficulties in the game by identifying sections with multiple nearby vertices because each vertex in the collapsed graph represents a major variation in the game state.

The experimental results show that the inter-cluster verification variant (IC) provides better results than the *DBSCAN* for collapsing similar segments in the graph. The statistical

analysis of both experiments shows that this is true for almost all the studied cases, including random and monotonic behavior and different graph types. The variable epsilon variant (VE), which is responsible for generating a customized epsilon for each cluster, is inferior to the IC variant when used alone. However, in the automatic experiments, the combination of both variants, resulting in the ICVE algorithm, provided better results when dealing with more randomized values. Unfortunately, the judges disliked the algorithm due to its sensitivity and thus it requires more refining before being used. Answering our research question, the IC algorithm showed to be the most effective algorithm for reducing the information while also preserving its overall semantics in all instances according to the judges and the automatic experiments. As a consequence, it has become the default option in Prov Viewer. However, all other algorithms are also available at the tool.

In the next chapter, we propose our last contribution for game provenance: an approach that merges multiple provenance graphs to allow the game designer to understand the causes that led a player to fail and another to succeed in the challenges presented by the game. Furthermore, our proposed approach can also be used as a debugging procedure to determine the reasons specific scientific trials failed to achieve the expected result while other succeeded.

CHAPTER 7 – FAULT LOCATION AND CORRECTION THROUGH PROVENANCE DIFF

7.1 INTRODUCTION

The wealth of provenance data collected during a game session is fundamental for understanding the mistakes made as well as reproducing the same results at a later moment. However, provenance data can be highly detailed and, depending on the game, can result in a huge quantity of tracked information leading to provenance graphs in the order of thousands of vertices. This wealth, but the huge quantity of data, can overwhelm the developer ability to analyze and understand the data, making more complex and tedious the process of identifying the reasons that may have caused a particular player to fail when compared with the results from other players.

In this chapter, we propose a provenance graph comparison approach for game analytics that identifies possible reasons and discrepancies that might have led the player to fail to reach the goal by contrasting with the performance of other players. We integrated our solution in our provenance visualization tool *Prov Viewer* and provide experimental studies using a projectile motion game simulation. The experiment shows that the proposed method is capable of identifying all the causes that could have led to a failure to reach the goals as well as proposing changes to make it work in the next execution.

The rest of the chapter is organized as follows: The Section 7.2 presents our approach and Section 7.3 presents the evaluation. Section 7.4 presents the related work in this area. Finally, Section 7.5 provides final considerations of this chapter.

7.2 PROV-DIFF

In this section, we discuss our approach for comparing provenance graphs, generated by different game sessions. Our approach uses a unified provenance graph to infer the probable causes that led to failure through graph differences. However, the process of creating a unified graph requires three elements: (1) a **matching heuristic** to match vertices from different graphs, (2) the definition of **vertex similarity**, and (3) a **graph merging algorithm**. Note that the merge process is done by merging two graphs at a time and, consequently, the *matching heuristic* uses only two graphs at a time as well.

In the following subsections, we discuss all these requirements to generate a unified graph. We also discuss our approach for comparing provenance graphs to determine the probable causes that led to the failure of a particular game session.

7.2.1 **MATCHING HEURISTIC**

A *matching heuristic* for vertex selection is used in order to restrict the search space for *vertex matching* and avoid making a Cartesian product between vertices from both graphs. Furthermore, the heuristic decides how the comparison between vertices from different graphs is made. It always chooses two vertices (one from each graph) to pass them to the *Vertex Similarity* algorithm for comparison.

The *matching heuristic* can simply restrict the search by neighborhood (*e.g.*, vertices from same agent type), or by specific vertex attributes (*e.g.*, time, coordinates), or both. For example, selecting vertices from one graph and vertices from the other graph that are temporally close and/or spatially nearby. Another example would be a heuristic that considers neighborhood, restricting the vertex selection to match only those that belong to the same type of agents, such as the player, non-playable characters, or a specific enemy type.

Currently, we use a simple heuristic that uses temporal information to compare vertices from provenance graphs generated from the same game (e.g., different players playing the game or multiple game sessions from the same player). The heuristic sorts all vertices from each graph by their temporal information and then proceeds to compare the first vertex from one graph with all the vertices from the second graph in an ascending order. If there is a match based on the *vertex similarity algorithm*, then the search ends and the matched vertices are merged. Both vertices are also added to a list of visited vertices to avoid matching them again. The heuristic then continues to the next vertex from the first graph and compare with the vertices from the second graph that does not belong to the list of visited vertices in an ascending order.

7.2.2 **VERTEX SIMILARITY**

The *Vertex Similarity* algorithm, also known as the distance metric function, always compares two vertices (*e.g.*, *v1* and *v2*) for the (similarity) evaluation to establish the similarity among them. The similarity value between two vertices ranges from 0 to 1, where 0 represents total mismatch (0%) and 1 represents a total match (100%). consider $G1 = (V,E)$ as a directed graph where $V = \{v_1, v_2, ..., v_n\} \mid v_i, 1 \leq i \leq n$ and consider $G2 = (V,E)$ as a directed graph where

$V = \{v_1, v_2, ..., v_m\} \mid v_i, 1 \leq i \leq m$. The comparison algorithm always compares vertices v_x and v_y at a time, where $v_x \in G_1$ and $v_y \in G_2$ and $G_1 <> G_2$. This process is divided into four steps: (1) **distance metric configuration**, (2) **vertex type verification**, (3) **attribute evaluation**, and (4) **similarity evaluation**. Furthermore, the comparison algorithm uses a *configuration file* where it contains a list of elements and each element contains an attribute name, a value that represents the acceptable *error margin* for that specific attribute, and the weight of that attribute for the similarity calculation. Figure 60 illustrates the overview of this entire process.

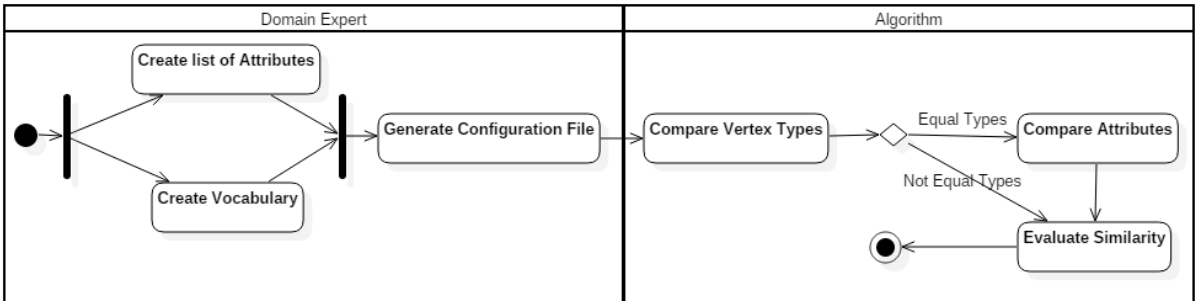


Figure 60: Overview of the Vertex Similarity process

The **first step** is the **distance metric configuration**. This step uses a *configuration file*, which is consulted during the third step (**attribute evaluation**) to determine if the attributes values from v_x and v_y can be considered similar when their difference is below the accepted *error margin*. Moreover, the algorithm uses a value that represents the *similarity threshold*, which must be informed by the user inside the *configuration file*. This *similarity threshold* is used by the last step (**similarity evaluation**) to determine when two vertices should be considered similar.

For example, [(*Timestamp*, 0.2, 5), (*HealthPoints* 5, 2)] would be an attribute list in the *configuration file* which says that the *Timestamp* attribute has a weight of 5 with an acceptable *error margin* of 0.2. Meanwhile, the *HealthPoints* attribute has a weight of 2 with an acceptable *error margin* of 5. Thus, specific attributes can have more impact on the similarity calculation, such as attributes that have *Time* and *HealthPoints* information could weight *n* times more than other attributes when determining the similarity factor. If the user does not provide this *configuration file*, then the *error margin* uses a default pre-defined value and all attributes will have the same weight.

Similarly, the user can also provide a *vocabulary* inside the *configuration file*, which lists all accepted "similar" string values. The *vocabulary* is a dictionary that is constructed through multiple lists of strings, such as "*House, Cabin, Cottage*" and "*House, Apartment*". It breaks each list into separate strings and adds them in the dictionary, using the string's name as

the key in the dictionary and the list as a value, which represents the accepted strings for the current string. Thus, using the example of "*House, Cabin, Cottage*", the *vocabulary* will have 3 entries in the dictionary (one for each string) as follows: {"*House*": ["*Cabin*", "*Cottage*"], "*Cabin*": ["*House*", "*Cottage*"], "*Cottage*": ["*House*", "*Cabin*"]}. However, if we add a new list in the *vocabulary* with "*House, Apartment*", then the vocabulary will update the *House* value to be ["*Cabin*", "*Cottage*", "*Apartment*"] (added *Apartment* in the accepted strings for *House*) and insert a new entry in the dictionary with "*Apartment*": ["*House*"]. If the user does not inform any list to the *vocabulary*, then the vocabulary will be empty and strings will only be considered similar if they are equal.

The **second step** from the *Vertex Similarity* algorithm is the **vertex type verification**. This step receives two vertices (v_x and v_y) from the *matching heuristic* and checks the type of v_x and v_y to verify if they match. If they belong to the same vertex type (*i.e.*, Agent, Activity, or Entity) then it proceeds to the third step (**attribute evaluation**), which evaluates the attributes from both vertices. In the case where the types are mismatched, then the vertices are not considered similar and the comparison is halted, setting a similarity factor of 0%, skipping the third step, and going directly to the fourth step.

The **third step** of the algorithm, which is the **attribute evaluation**, tries to match each attribute from one vertex (v_x) with an attribute from the other vertex (v_y), comparing their values. Thus, the algorithm goes through all attributes from v_x and tries to find the same attributes in v_y using the attribute's name. If v_y has an attribute with the same name as v_x*'s* attribute, then it compares their values to determine whether both vertices have similar values. This comparison also searches the *configuration file* to verify the acceptable *error margin* for the attribute when dealing with numeric values. Thus, if the difference between the numeric values is lower than the accepted *error margin*, then they are considered to be similar values. For strings, the algorithm searches the *vocabulary* using the string value from v_x to determine if the string value from v_y is similar. If the string from v_x is in the *vocabulary* and it contains v_y's value as accepted strings, then they can be considered similar. If v_y has an attribute that was not matched with v_x, then this attribute from v_y is added to the similarity evaluation as a zero match because v_y has attributes that v_x does not have, thus they can never be considered as a total match. The process is analogous if v_x has an attribute that was not matched with v_y.

For example, consider that the attribute being evaluated is *Time* and v_x*'s* is equal to 5 and v_y*'s* is equal to 5.1. If we use the [(*Timestamp*, 0.2, 5), (*HealthPoints*, 5, 2)] attribute list,

then the difference between v_x and v_y (5 - 5.1 = 0.1) is below the accepted variance (0.2 for *Timestamp*).

The **fourth step**, which is the **similarity evaluation**, determines if v_x and v_y can be considered similar and thus suitable for combining into a single vertex in the unified graph. The similarity factor, which is used for the evaluation, is calculated from the number of attributes that were considered similar in both vertices during the second step. This number of matched attributes is then divided by the total number of distinct attributes from both vertices, generating a value ranging from 0 to 1, where 0 represents a total mismatch (not a single attribute could be matched either by name or by value) and 1 represents a total match (all attributes from v_x are in v_y with similar values and v_y has no attribute that v_x do not have and vice versa). The resulting number from this division is the *similarity factor*. The *similarity factor* is then compared with the *similarity threshold*. If the *similarity factor* is below the accepted *similarity threshold*, then v_x and v_y are not considered similar. However, if the *similarity factor* is equal or greater than the *similarity threshold*, then *both* vertices can be considered similar vertices. Note that these two vertices are the ones received from a *matching heuristic* during the step two. Furthermore, this entire process is only to inform if these two received vertices can be considered similar.

For example, if the *similarity threshold* is 1, then vertices will only be considered similar if all their attributes and respective values match, which would result in a similarity value between v_x and v_y of 1. Since the similarity value equals 1 and the *similarity threshold* is set to 1, then these two vertices can be considered similar (similarity value is equal or greater than the similarity threshold). However, if we use a *similarity threshold* of 0.75 (which translates as 75% match) then v_x and v_y would be considered similar if at least three-quarters of all their attributes match (considered equals or similar).

7.2.3 **GRAPH MERGE**

After establishing that two vertices are similar, then the next step is to merge them to create a new vertex that represents the two original vertices. This newly created vertex will belong to the unified graph, while the original vertices will not be a part of it. The merge process creates a new vertex of the same type of the original vertices (*i.e.*, agent, activity, or entity) and has all attributes from both vertices and their respective values plus a new attribute showing the graph it originally belonged. For numerical attribute values, it shows the average for that attribute and the minimum and maximum values in the following notation: *average_value(minimum_value ~ maximum_value)*. For String values, it shows all different

values separated by a comma (*e.g.*, String_Value1, *String*_Value2, String_Value3). A new vertex ID is created to represent the merged vertex. Figure 61 shows an example of displayed data from a merged vertex. The merged vertex was generated from the vertices shown in Figure 62 (even though they have the same ID, they belong to different graphs). We use the five-number summary (minimum, 1st quartile, median, 3rd quartile, maximum) when there are more than two vertices merged together into a single summarized vertex. This only occurs when merging multiple graphs.

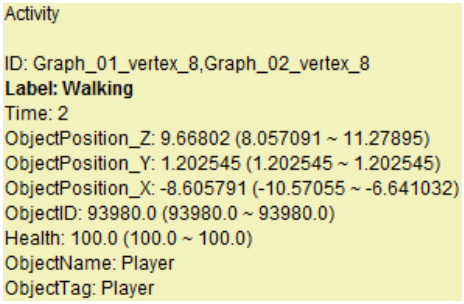


Figure 61: Fragment of a Merged Vertex

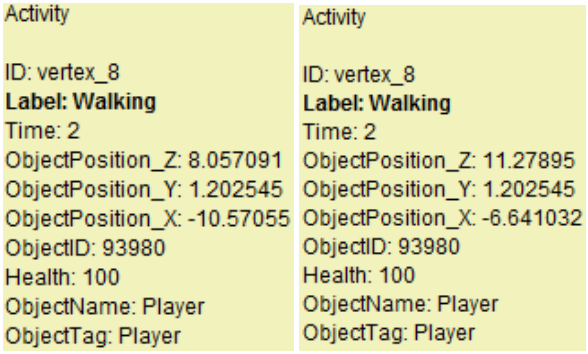


Figure 62: Original vertices from Figure 61

After inserting the vertices in the unified graph, it is also necessary to update all edges that used any of the old vertices for generating the merged vertex. This update changes the edge's source or target that points to either vertex that was used for the merge to point to the new generated (merged) vertex. Otherwise, the edge would point to an invalid vertex since the original vertices are not in the unified graph. Furthermore, when an edge had its source and target vertices updated, then it is necessary to check if there is another similar edge with the same type that also had its source and target updated (possibly from the other graph). When two or more edges have the same type, source, and target vertex, then they can be merged into a single edge. The merged edge's value will be the weighted average value from all edges used in the merge and have its weight increased (the default weight is one) to reflect the number of edges used in the merge. Therefore, if we merge two edges with values 2 and 4, the resulting

merged edge would have a weight of 2 and a value of 3 (since they had the same weight). However, if later we merge this new edge with a third edge that has a value 6, then the new merged edge would have a weight of 3 (two from the first merge and one from the third edge) and a value 4 due to the different weights. If we merge this new edge (value 4, weight 3) with another merged edge (value 14, weight 2), then the resulting edge would have a value of 8 $((4 \times 3 + 14 \times 2)/(3 + 2))$ and weight of 5.

Below we present an example of a unified graph generated from two graphs. These graphs were generated from the *Angry Bots* game and were rendered with our tool *Prov Viewer*. Figure 63 shows the first graph used for the summarization process, while Figure 64 shows the second graph. Figure 65 shows the game session by showing the player's initial location, the locations of the enemies, and the sequence of sections explored in the map. The player (red circle) starts at the bottom of the map and then explores the map passing through the first area (orange rectangle). Then, the player goes to the second area (yellow rectangle) and finishes at the last area (blue rectangle). There are two enemies in the second area (pink circles). Green vertices represent the player actions, while pink vertices represent the enemy actions.

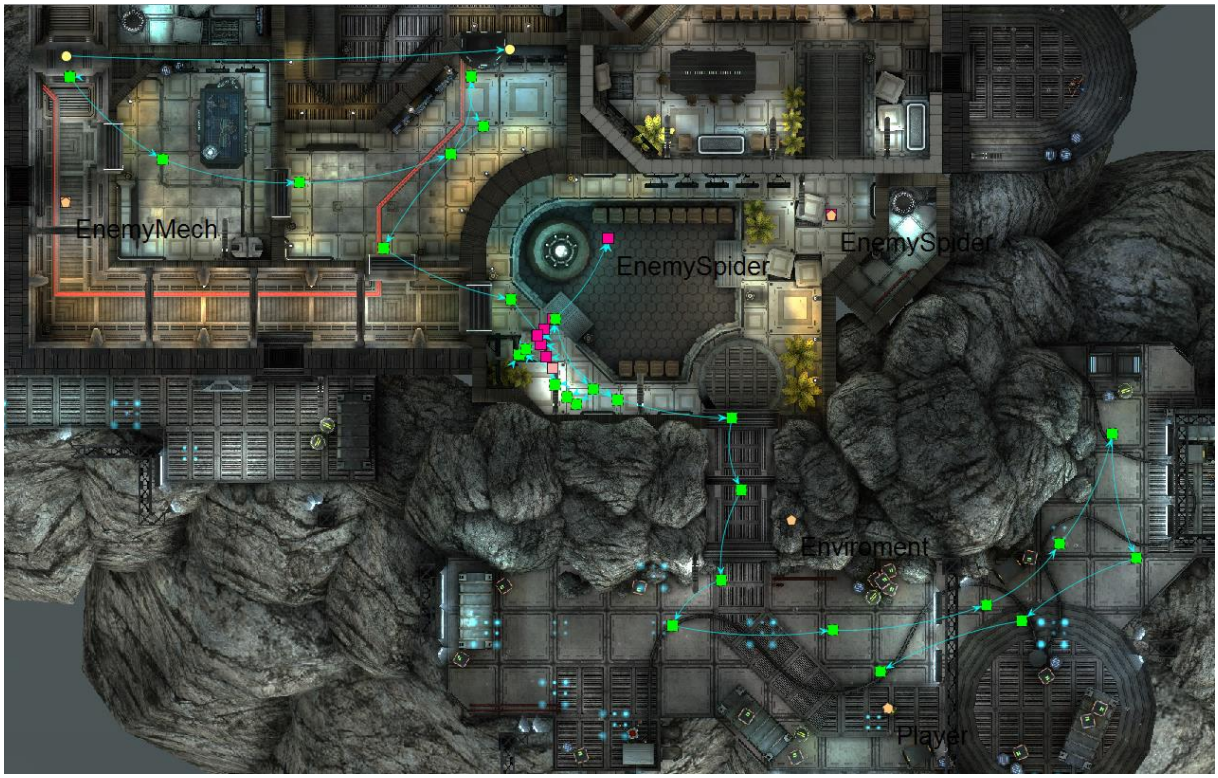


Figure 63: Original Graph #1 used for the summarization

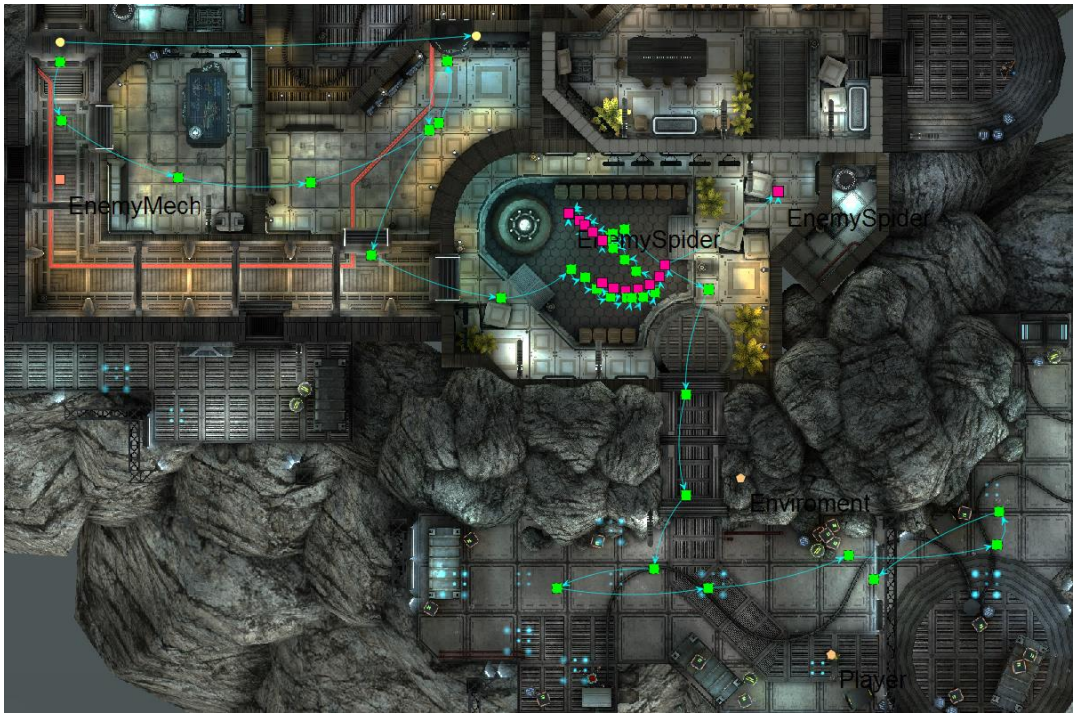


Figure 64: Original Graph #2 used for the summarization

Figure 65: Game layout (based on the graph from Figure 63).

We can see that both graphs (Figure 63 and Figure 64) have some variations, including enemy engagement (in graph #1 the player only engaged 1 enemy, while in graph 2 the player engaged both) and actions (vertex) location, which is based on specific attributes of the vertex. In this case, we are referring to enemy engagement a situation when vertices from different agents (the player's green vertices and the enemy's pink vertices) have some kind of

relationship. Figure 66 illustrates an engagement example. Figure 67 shows the unified graph generated based on graphs from Figure 63 and Figure 64. This summarization example is based on temporal and spatial information (coordinates) to determine vertex similarity. By analyzing carefully, the merged vertices (grey) have slightly different positions from the original vertices because these vertices use the average values for the spatial coordinates of the merged vertices (*e.g.*, Figure 61). However, we are planning to extend this feature (vertex position) to allow the user to choose between different functions to determine the final spatial coordinate, such as choosing the minimum value, the maximum value, the average, mode, or the median value from the summarized vertex.

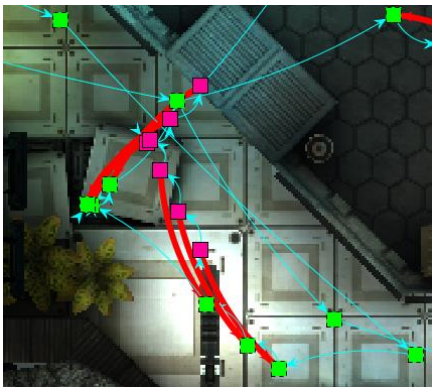


Figure 66: Enlarged section of the graph illustrating an enemy engagement. Note the red edges that link a pink vertex (enemy) with a green vertex (player). These red edges (previously hidden to not overwhelm the graph presentation) represent damage taken.

Figure 67: Unified graph using a matching heuristic with a threshold of 0.9 and having only margins of error for spatial and temporal information. Merged vertices had their color altered to have blue color for an easier identification in the default visualization scheme.

The presented merge approach does not trivially support an inverse operation since the original vertices and edges are discarded/updated and a newly merged vertex is created in their

place. However, *Prov Viewer* also supports another type of graph merge that allows the user to undo the merge. This second type of graph merges actually preserves all the vertices and edges from both graphs by creating a cluster with the vertices (one from each graph) that were considered similar. Thus, the user can undo the merge by simply expanding the cluster. The downside of this approach is the fact that it preserves all the original data (vertices and edges) inside the cluster-vertex and needs to create new data to represent the cluster-vertex and new edges that connect the newly formed cluster-vertex while preserving the old edges for the undo operation. This new information grows linearly when merging multiple graphs since the merge process is done by merging two graphs at a time. Thus, a cluster-vertex can have a cluster-vertex inside it and another cluster-vertex inside the cluster-vertex and so on. Every time the user expands a cluster-vertex from the merge operation, then it is undoing the last merge that affected that particular vertex that is being expanded.

7.2.4 **DETECTING GRAPH DIFFERENCES**

Comparing the differences between two or more graphs becomes trivial after generating the unified graph since each vertex contains information related to their origins. Thus, we can see all vertices from a specific graph and know how it differentiates from the other graphs inside the unified graph by comparing each vertex's origin. Furthermore, the original attributes values and their source are preserved when merging similar vertices. The graph diff can be done within the *Prov Viewer* by selecting a vertex color scheme that is based on the created attribute that states the origins of the vertex.

This visualization from *Prov Viewer*, when using for differentiating graphs in the unified graph, paints all the vertices that belong only to the selected graph in green, all vertices that do not belong to the desired graph in red, and all vertices that belong to more than one graph (including the desired graph) in grey. This allows for quick identification of segments that differentiates a particular graph from the others. Figure 68 shows the colored vertices from the unified graph that belongs to the first graph (from Figure 63) and Figure 69 shows a comparison between graphs in a zoomed section from the game.

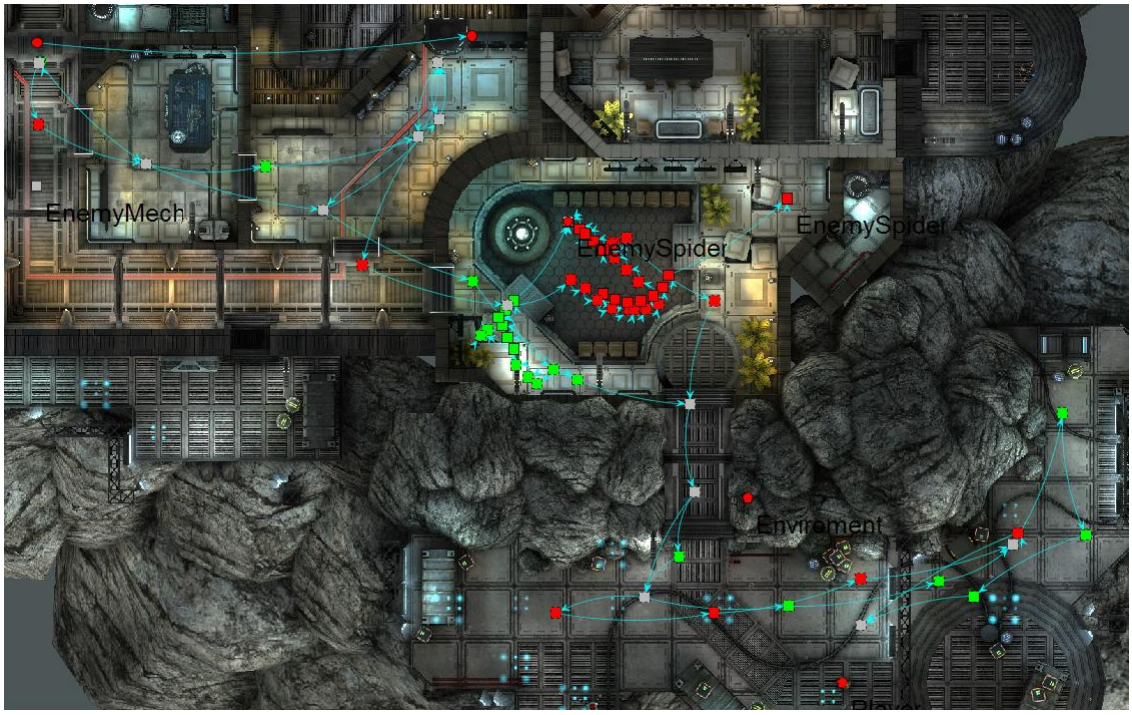


Figure 68: Unified graph from Figure 67 showing all vertices that belong only in Graph #1 (green vertices), vertices that belong only in Graph #2 (red vertices), and vertices that belong in more than one graph, including Graph #1 (grey vertices)

Figure 69: Comparison between graphs: Graph #1, Graph #2, and Unified Graph.

7.2.5 **COMPARING GAME SESSIONS THROUGH DIFFS**

The comparison process uses the unified graph to detect the causes of a problem and to infer possible solutions through graph diffs. The comparison algorithm compares the *current graph* (CG) that had negative results or failed to achieve the goal with all other graphs that had positive results in order to find the discrepancies that could have led to the failure. It searches the unified graph for the *shortest diff distance to a positive result graph* (SDDPG) and uses it as a baseline to determine the causes that led to the negative results through a vertex-by-vertex comparison. The vertices that only belongs to CG represent the causes that might have led to the failure. Meanwhile, the vertices that appear only in the SDDPG is the suggested **patch operation** that contains the changes that need to be made in order to reach the goal.

The CG might be able to reach the desired goal if it replaces the vertices that belong only to it by the ones that belong only to the SDDPG. As a consequence, this has potential to make the CG identical to the SDDPG, which is known to be able to reach the desired goal. However, the CG will only be a clone of the SDDPG if the unified graph used a similarity function that only considers vertices to be similar if they are 100% equals in attributes and values, possibly leading to an extremely large **patch operation**. In other cases, the CG will have vertices that were considered similar to vertices in the SDDPG but do not necessarily have the same content.

Figure 70 illustrates an example of our comparison algorithm inside *Prov Viewer*. *Grey vertices* represent the vertices that belong to both CG and SDDPG. These vertices were considered "correct" by the comparison algorithm because they appear in both graphs. *Red vertices* represent vertices that appeared in CG but not in SDDPG. These are the vertices that might be the reason for not reaching the goal. *Green vertices* represent vertices that appear in SDDPG but does not appear in CG. These vertices are the suggested **patch operation** to make CG reach the goal.

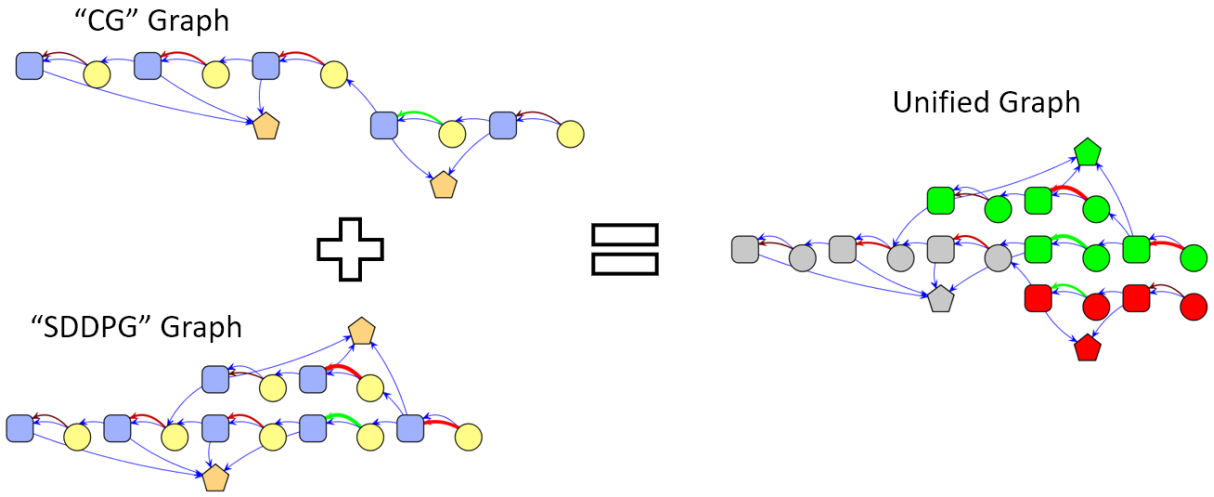


Figure 70: Comparison visualization using the unified graph inside *Prov Viewer*

Our comparison algorithm, when used outside the games domain, can be seen as a *provenance-based debugger* to debug experimental trials. In this scenario, it can determine why a specific trial failed while another had positive results by using the same process and logic described in this section. Our approach would detect the differences between trials, the possible causes that led the trial to fail (vertices only belonging to CG) and the proposed fix (**patch operation**) to make the trial reach the desired goal.

7.3 EVALUATION

In this section, we assess the proposed comparison algorithm for provenance graphs by determining the dependent variables *accuracy*, *retention*, and *harmonic mean* of our comparison algorithm. The *accuracy* metric tells us how many times our algorithm correctly predicted the probable causes that led to failing the goal. This is measured by applying the **patch operation** on the CG and verifying if the injected changes were effective to reach the goal. If the changes proved to be successful, then our algorithm correctly predicted the causes that led to the failure. The higher the *accuracy*, the better.

The *retention* metric tells us how many vertices remained unaffected by the **patch operation**. This metric measure how much of the CG was preserved and allow us to compare the algorithm accuracy based on the number of changes in the **patch operation**. The higher the *retention*, the better. Finally, the *harmonic mean* tells us the overall performance of the algorithm based on the compromise of accuracy and retention metrics.

7.3.1 MATERIALS AND METHOD

This experiment was executed using a shooting competition game simulation that uses projectile motion physics with the goal of hitting a target. We evaluated the accuracy of our comparison algorithm in detecting the probable causes that led to missing the target.

The simulation has nine configurable parameters: (1) bullet mass, (2) air density based on temperature, (3) air density based on altitude, (4) air drag, (5) initial X position, (6) initial Y position, (7) bullet speed on X-axis, (8) bullet speed on Y axis, and (9) target position. Moreover, it has one constant: gravity. The physics behind this simulation is described in Equation 2, where ρ is the air density, C_d is the drag coefficient, and A is the cross-sectional area of the projectile. The simulation goal is for the shooter to hit the target.

Equation 2: Projectile motion equations

$$\ddot{x} = -\beta \dot{x} \sqrt{\dot{x}^2 + \dot{y}^2}$$

$$\ddot{y} = -g - \beta \dot{y} \sqrt{\dot{x}^2 + \dot{y}^2}$$

$$\beta = \frac{\alpha}{m}$$

$$\alpha = \frac{\rho C_d A}{2}$$

We simulated a shooting competition with 15 participants. The competition is comprised of 20 rounds, wherein each round all participants have one shot to hit the target. The target position changes after each round. Each shot taken generates a provenance graph with the parameters used in the projectile motion simulation. Figure 71 shows an example of the generated provenance graph for this type of simulation, where entities in the graph represent the input parameters and activities represent the bullet position over time. The input parameters were generated randomly using a Gaussian distribution to simulate different players participating in the competition. We consulted real data values to select the mean for the Gaussian function and generate the sigma for the Gaussian distribution. As a result, all input variables were continuous.

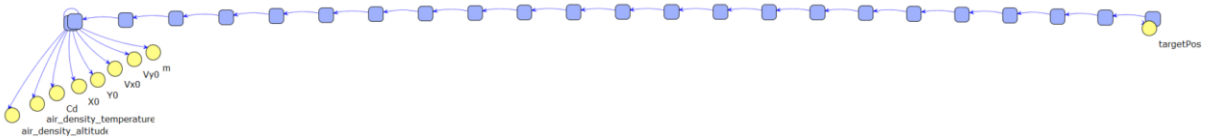


Figure 71: Example of provenance graph for the projectile simulation

For this experiment, we only consider the input variables as the probable cause of determining if the goal was reached or not, since the bullet final position is directly related to the nine input values described earlier. Thus, for comparison purposes, only the entities in the graph that represent input variables are taken into consideration by our algorithm, since all other vertices from the graph do not influence the final result.

Lastly, the results from our comparison algorithm are directly related to the unified graph, which requires a *similarity distance metric* to determine if two graphs are similar and calculate the DIFFs. All variables in this experiment belong to the continuous space, meaning that there will never be two equal values in the entire dataset. Moreover, the *similarity distance metric* is directly related to the *retention* metric because a value from CG can be considered similar to its counterpart from SDDPG even though they are not exactly equal and, thus, preserving the CG vertex. Thus, the **patch operation** will not change the vertex since it is equivalent to the one from SDDPG.

As such, the experiment execution plan was divided into five stages: (1) generate the dataset, (2) create different similarity coefficients, (3) generate the unified graph for each similarity function from stage 2, (4) execute the experiment using the unified graphs from stage 3, and (5) analyze the results. The entire simulation resulted in 300 graphs since it had 15 participants and 20 rounds, totaling 300 shots. From these 300 generated graphs, only 16 graphs

managed to hit the target, which is equivalent to 5.33% of the shots reaching the goal of the simulation (hit the target). Our comparison algorithm requires at least one graph to have reached the goal and this restriction was satisfied in the generated dataset.

The second stage is responsible for generating the *similarity distance metric* for the experiment. We created 10 different metrics. Each *similarity distance metric* uses the same factor of standard deviation (sigma) to define the similarity threshold for each input variable. The difference between the *similarity distance metrics* is the factor used for the thresholds. The generated *similarity distance metrics* are: (1) 0-Sigma, (2) 0.25-Sigma, (3) 0.5-Sigma, (4) 0.75-Sigma, (5) 1.0-Sigma, (6) 1.25-Sigma, (7) 1.5-Sigma, (8) 1.75-Sigma, (9) 2.0-Sigma, (10) 3.0-Sigma. Thus, the first metric (0-Sigma) will only consider two vertices to be similar if their attributes' values are within zero standard deviations apart, or in other words, if their numeric values are the same. We did not add more points between 2.0-Sigma and 3.0-Sigma due to the rule "68-95-99.7", which states that, for a normal distribution, 68% of the values fall between one Sigma around the mean, 95% fall between two Sigma around the mean, and 99.7% fall between three Sigma around the mean. Thus, the difference between 2-Sigma and 3-Sigma is only 4.7%, which is too small to generate any significant impact on the result. Nevertheless, we kept 2-Sigma and 3-Sigma in the experiment to show that the difference when using these metrics is not significant.

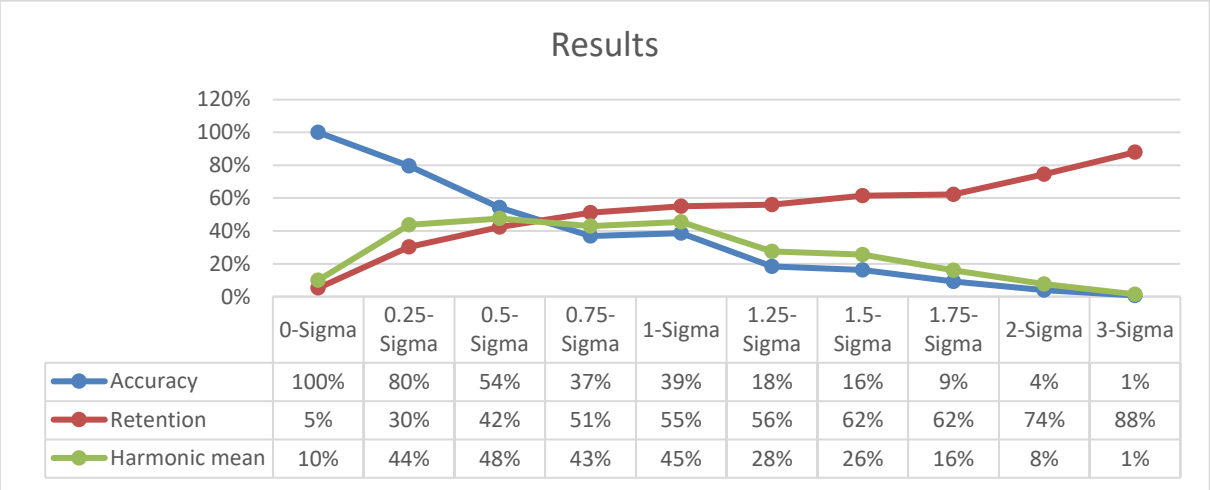
The third stage is responsible for creating the unified graphs using the *similarity distance metrics* from the second stage and the 300 graphs from stage 1. Thus, at the end of this stage, we had 10 different unified graphs that represent different *similarity distance metrics* when applied in the same 300 graphs.

Finally, we executed the experiment using the unified graphs from stage 3. We calculated the *accuracy*, *retention*, and *harmonic mean* metrics for each unified graph. This process is done by applying the **patch operation** in each graph that had not reached the goal (284 graphs from 300) for each one of the unified graphs. We re-ran the projectile motion simulation in each patched graph to determine if the modifications were sufficient to allow the shot to hit the target.

7.3.2 **RESULTS AND DISCUSSION**

The graphic from Figure 72 shows the obtained results from our evaluation for each one of the *similarity distance metric* used to generate the unified graph. The *retention* rate was calculated for all graphs, independently whether the patch worked or not.

These results show that our comparison algorithm can achieve a 100% accuracy rate. However, this only occurred when using the 0-sigma similarity, which has the lowest retention rate of 5%. This means that the algorithm basically discarded the failed graph and replaced it by a clone from a graph that reached the goal. This was done in order to make the failed graph work since all vertices are considered to be different due to the zero tolerance for similarity in a continuous domain. Considering that the SDDPG reached the goal, then it is reasonable that its clone will also reach it.



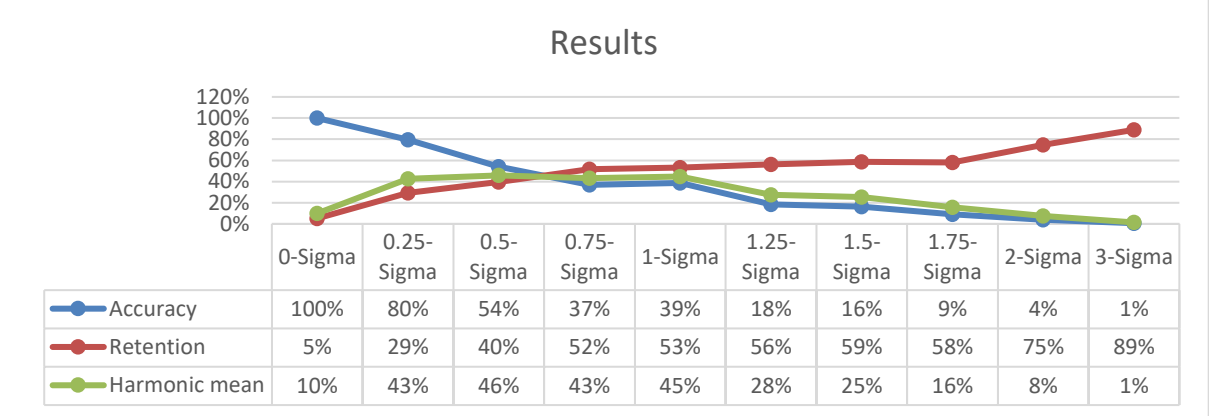
	0-Sigma	0.25-Sigma	0.5-Sigma	0.75-Sigma	1-Sigma	1.25-Sigma	1.5-Sigma	1.75-Sigma	2-Sigma	3-Sigma
Accuracy	100%	80%	54%	37%	39%	18%	16%	9%	4%	1%
Retention	5%	30%	42%	51%	55%	56%	62%	62%	74%	88%
Harmonic mean	10%	44%	48%	43%	45%	28%	26%	16%	8%	1%

Figure 72: Evaluation results showing Accuracy, Retention, and harmonic mean metrics for each similarity coefficient used.

The results from other *similarity distance metrics* show that the *accuracy* and *retention* metrics are inversely proportional: *accuracy* decreases as *retention* rate increases. This also sounds natural since if we preserve the failed graph, then the goal will not be reached. Furthermore, the more elements we preserve from the failed graph, the lower the chances are of identifying the reasons and fixing it since the problem can be bigger than what it is allowed to be changed by the algorithm. Nevertheless, this allows the user to choose the desired *retention* rate of the analyzed graph at the cost of losing *accuracy* of the prediction based on the patch size.

Thus, it is up to the user to define whether the *accuracy* or *retention* is more important. However, by looking at the obtained results, it is not recommended to have *retention* rate greater than 55% because otherwise, the average *accuracy* rate drops from 40% to 20%. The Harmonic mean metric can be used in cases where the user is after the overall performance of the algorithm based on the compromise of *accuracy* and *retention* metrics. In this case, the optimal *similarity distance metric* is 0.5-Sigma, resulting in 55% *accuracy* and 40% *retention* rate.

The *retention* rate changes to the values illustrated in Figure 73 if we consider only the graphs that were able to reach the goal after the **patch operation**. These results are similar to those presented in Figure 72. However, the average *retention* rate is a little lower when considering only these graphs. Nevertheless, the difference is minimal and does not change nor heavily impact in the overall results.



	0-Sigma	0.25-Sigma	0.5-Sigma	0.75-Sigma	1-Sigma	1.25-Sigma	1.5-Sigma	1.75-Sigma	2-Sigma	3-Sigma
Accuracy	100%	80%	54%	37%	39%	18%	16%	9%	4%	1%
Retention	5%	29%	40%	52%	53%	56%	59%	58%	75%	89%
Harmonic mean	10%	43%	46%	43%	45%	28%	25%	16%	8%	1%

Figure 73: Evaluation results using retention rate of graphs that were correctly debugged.

7.3.3 **THREATS TO VALIDITY**

We identified internal and external factors that may influence the results. In relation to internal validity, we compute the algorithm's *accuracy* from the result of the **patch operation** by verifying if the modified graph is able to reach the goal. Regarding external validity, we mitigated sample bias by randomly generating 300 different graphs using Gaussian distribution and having a set with more than one graph that reached the goal.

7.4 RELATED WORK

Related works were selected from those related to provenance reproducibility since the majority of comparison approaches for games rely solely on artisanal methods of exploratory analysis on tracked data to try to understand what went wrong. However, workflow management systems (AFGAN et al., 2016; ALTINTAS et al., 2004; CALLAHAN et al., 2006; DAVISON, 2012; HIDEN et al., 2013; HULL et al., 2006; KIM et al., 2008a) are able to keep track of provenance traces and their evolution through version control. Unfortunately, simply comparing through hash-code, code, labels, or file names is not enough since different data can have similar metadata, causing false-positives when computing the diff between provenance traces.

The *PDiffView* approach (BAO et al., 2009) detects the difference between workflow executions, focusing on finding the edit distance between two valid workflow runs that have the same specification. However, this approach only considers the workflow structure and ignores details that are used during scientific analysis of the workflow.

The SGProv approach (EL-JAICK; MATTOSO; LIMA, 2014) creates a summarized provenance graph to reduce redundancy by avoid duplication of similar nodes when merging different graphs. However, this process is not generic and only considers activities that belong to the type of "program". Furthermore, the summarization process only compares the type and name of the "program" node (vertex) and ignore the metadata contained within the vertex.

Another recent approach is *why-diff* (THAVASIMANI; CAŁA; MISSIER, 2017), which is capable of explaining the differences between two workflow runs by looking at the execution provenance associated with the workflow. This approach can find the divergence points to trace the causes that can explain the output differences and provide evidence that the detected changes had impacted the output. The main advantage of this approach is that it is the only approach that looks at the metadata within a vertex. The metadata used is the configuration property of the activity nodes captured by eScience Central, which contains a relation of files and libraries used by the activity. This can be similarly compared to attributes within an activity vertex. However, the *why-diff* approach has no method of determining similarity context besides perfect matching and only detect the differences in file names in the properties. Moreover, the authors only consider isomorphic provenance graphs and do not appear to work when comparing graphs with slightly different structures due to a lack of a matching heuristic.

7.5 FINAL CONSIDERATIONS

This chapter proposed a novel approach for comparison of provenance graphs. Currently, only one existing approach considers vertex metadata during the diff operation. However, that approach only considers perfect matching of the metadata information and only work on isomorphic graphs. Meanwhile, our proposal provides a similarity algorithm to determine if two distinct metadata can be considered similar, even when having minor divergences and also having a matching heuristic to compare graphs that are not isomorphic.

Our experimental results show that our comparison approach is capable of accurately detecting underlying issues that could have led to failure by comparing the provenance graph with another provenance graph that was known to reach the goal. Moreover, the results show that our algorithm can work with a flexible definition of similarity. However, this flexibility for

similarity definition directly impacts in our algorithm's accuracy. This, in a logical point of view, makes sense since we are broadening the definition of similar values and, as a result, increasing the acceptable error margins in the interpretation of what is considered similar objects, which can lead to errors when saying that two objects are similar even when they are actually completely different.

This chapter provided the last contribution to our work. Thus, in the next chapter, we conclude this thesis, listing our contributions, limitations of our approach and future work in the new fields of the study presented throughout this work.

CHAPTER 8 – CONCLUSION

8.1 CONTRIBUTIONS

In this thesis we presented a novel approach for tracking and displaying the game provenance data, opening a new research area for Game Analytics for tracking telemetry data and extracting knowledge from it. The richness and completeness of the provenance data allows for more abstract analysis over tracked game data, broadening the possibilities for different types of analysis and applications for tracked data. The game provenance data is a structured and directed acyclic graph, also known as a provenance graph, that represents the entirety of the game session. This graph can be used for data mining techniques to extract knowledge or used for exploratory analysis by generating a visual representation of the data through a dynamic and interactive node-link diagram. The proposed approach also supports the analysis of multiple game sessions by merging different provenance data to generate a summarized provenance graph that can be used to understand the reasons that led to different outcomes by detecting the differences between game sessions and displaying the probable causes.

The advances presented in this thesis can be divided in four major contributions: (1) The *PinG for Unity* (*PinGU*) framework, for tracking provenance data, (2) *Prov Viewer*, for interactive display and visualization of provenance graphs, (3) three distinct summarization techniques, for reducing the overall provenance graph size with minimal information loss, and (4) a comparison technique for provenance graphs, to discover underlying issues that might have affected the result.

In a previous work, we observed the positive impacts that provenance data have on understanding the events in a game session by using a conceptual framework with tracking and visualization prototypes. Thus, we created the PinGU, a framework for collecting provenance data from a game session. Our component facilitates the process of tracking and storing provenance for data exploration and analysis and unlike its predecessor, it is no longer a conceptual framework; it actually provides implementations for tracking and automatic management of provenance data. Designers only need to import and incorporate PinGU in their projects and create domain-specific tracking methods in the desired actions and events. The generated provenance graph from PinGU allows for post-game analysis to discover issues that contributed to specific results in the game session. This analysis can be used to improve the understanding of the game session and to identify actions that influenced the outcome. The

graph can also be used to analyze the game story development, how it was generated, and which events affected it.

Using provenances graphs to represent a game session brings the challenge of displaying this provenance data to the game designers and analysts. Thus, we also proposed a visualization solution focused on displaying and interacting with provenance graphs, allowing designers to perform exploratory visual analysis over the collected game session data. Our solution was implemented as a provenance visualization tool named *Prov Viewer*, which uses graphics and interactive features to distinguish information for comprehending the events. These features affect the displayed graph by transforming vertices and edges, changing their shapes and color according to the information type. Another important feature is information filter, which omits displayed information that is not relevant to the analysis. This filtering is important for analysis because it reduces the amount of displayed information to those that are the focus of interest of the user. *Prov Viewer* also has other graph manipulation features, including but not limited to, manual and automatic collapses, graph merge, and graph comparison techniques.

Although our visualization tool offers many features to interact with the provenance graph, there was still a problem when visualizing provenance graphs from long game sessions. Thus, we proposed three automatic data filtering techniques for provenance graphs, based on *DBSCAN*, an existing density clustering algorithm. Our algorithms take into consideration the temporal sequence of information of a provenance graph to summarize tracked data to a more manageable size through the usage of collapse strategies. In the game domain, vertices that were not collapsed tend to represent drastic changes in the game state and might be worth displaying during the analysis, while all collapsed vertices tend to represent minor variances around the state. We evaluated our approach for collapsing provenance graphs through two different experiments using automatic evaluation techniques and experts. The experimental results showed that at least one of our algorithms (*IC* variant) provided better results than the *DBSCAN* algorithm for collapsing similar segments in the provenance graph. From the automatic experiments, the *IC* variant proved to be better than *DBSCAN* in all cases, while in the expert experiments it had a minimum of 11 votes in all cases. Our *IC* variant was only surpassed by *DBSCAN* in one case (R-Tree), which *DBSCAN* had 15 votes against 11 votes from *IC*.

Lastly, a key usage of provenance is to understand how an outcome was reached. As a final contribution, we proposed a provenance graph merge and comparison approach. Our proposed approach can identify possible reasons and discrepancies in a provenance graph that

might have led a player to fail to reach the goal. To do so, we contrast the provenance of the game session that failed with the combined provenance of all successful game sessions. We integrated our solution in our provenance visualization tool *Prov Viewer* and provided an experimental study using a projectile motion simulation. Our experimental results showed that our comparison approach is capable of detecting underlying issues that could have led to failure by comparing with another provenance graph that was known to reach the goal. Furthermore, the results showed that using the optimal *similarity distance metric* results in 55% *accuracy* and 40% *retention* rate but can reach as far as 100% *accuracy* when having a *retention* rate close to 5%.

While the main application of provenance in this work is for games, we believe that the concepts discussed throughout this thesis are applicable to other domains and might be useful to support advanced forms of analysis. The proposed concepts could be applied in scientific experiments to visualize experiment provenance data, to debug the experiment in order to identify issues and better understand the obtained results.

Lastly, Table 9 extends the comparative chart among approaches in the games domain from Section 2.7 of Chapter 2. This table adds our proposed approach, which is our PinG for Unity framework and *Prov Viewer*. The table compares our approach with the previously discussed approaches from that chapter.

As can be seen in the table, our approach stands out from the rest by tracking richer game data, including contextual information from actions and events. Our approach also offers methods to mitigate information overload from huge game sessions and allow to compare multiple sessions for analysis to determine possible causes that led to certain outcomes. Lastly, our approach is the only approach that records the causal relationships (cause-effect) from the events in a game session.

8.2 LIMITATIONS

Although we invested the effort to conceive and build a useful and functional provenance infrastructure for games, our work has some limitations. Our provenance gathering framework requires manual instantiation of existing functions to inform, for each action, all possible causal relationships that can happen during the game. This information is usually available in the game design document. However, some unforeseen relationships could happen in a game due to the interaction of different complex mechanics or features. In the event that this occurs, our gathering framework would not be able to detect and capture these unforeseen causal

Table 9: Comparative chart of approaches

Features	GVM (JOSLIN; BROWN; DRENNAN, 2007)	TRUE (KIM et al., 2008b)	Playtracer (ANDERSEN et al., 2010)	Play-Graph (WALLNER, 2013)	PinG framework + Prov Viewer
Graph	x	x	√	√	√
Graphic	√	√	x	x	x
Event Context	x	√	x	x	√
Actions	x	√	x	√	√
Statistical Data Analysis	√	√	□	□	□
Cause-Effect	x	x	x	x	√
Temporal Information	√	√	x	√	√
Spatial Information	x	√	x	√	√
Information Overload Mitigation	x	x	√	√	√
Multi-session analysis	x	x	√	√	√

relationships and would need to be inferred by the game designers when analyzing the provenance data.

A limitation of *Prov Viewer* is related to scalability, regarding processing performance. Algorithms in *Prov Viewer* are not fully optimized for manipulating graphs with multiple thousands of vertices and edges, although we developed a level of detail method. Thus, its performance may degrade when dealing with huge graph sizes. Another limitation is based on input data. *Prov Viewer* can interpret PROV-N. However, PROV-N can only be used as an input and not output when exporting the graph. All operations done within the tool are lost when exporting to the PROV-N format due to format restrictions, resulting in exporting the original graph without any collapses or merge data. Nevertheless, the tool's own XML format support all the features it uses.

Regarding our similarity algorithm to filter sequential data, they require parameter tuning similar to any other clustering technique. A way to avoid meticulous parameter tuning is by

using factors of standard deviations as the threshold value. Moreover, since they were based on the DBSCAN algorithm, they inherited the non-deterministic nature.

Lastly, a limitation of our comparison approach is that it requires at least one graph that reached the goal, otherwise it would not work. Furthermore, the effectiveness of the algorithm is directly linked to the unified graph and the definition of similarity distance metric.

8.3 FUTURE WORK

There are some opportunities for future work to improve any of the proposals in this thesis or even expand with new studies due to the richness of provenance data. Our PinG concrete framework can be improved to automate even further the data tracking, especially for influences, which is the most complex aspect of the provenance tracking. Our component could also be extended to include other game engines, such as Unreal® Engine, due to their recent business change for indie developers.

Moreover, due to the quantity of the data extracted by PinG, we recommend studying other techniques to improve even further the visual analysis process. These studies involve, but are not limited to, automatic graph inferences, data mining, graph reduction, multiple graph analysis to compare multiple game sessions or even cycles during a game (*e.g.*, laps in a racing game), enabling better strategies of provenance gathering that take advantage of the game's genre.

Other possible future work, related to information visualization, include more elaborate algorithms to analyze the provenance data and suggest which information can be omitted to reduce the graph to acceptable sizes; more types of graph visualization techniques; and more graph layouts, including a support for dynamically loading new layouts in the tool at run-time, even handcrafted ones.

In addition, we also recommend future work related to provenance graph summarization in the aspects of further refinement of the heuristics and resulting algorithm, especially the VE heuristic. Other possible future work is to further refine our strategy to run the automatic experiment for evaluating graph summarization. Our initial result from both experiments (automatic and experts) shows an alignment of algorithm selection, which can be an indication of its possible effectiveness when dealing with graphs that have similar behavior of those we used, which can be very common across multiple domains.

There are other possible future works in relation to comparing provenance graphs. Our comparison only considers the closest graph that reached the goal of determining the causes that could have led to failure and proposing a new solution. A possible future work is related to finding good patterns from graphs that reached their goals to improve the chances of reaching the same goal in future iterations. Similarly, another approach could detect bad patterns that should be avoided. Another work could use statistical analysis from previous executions to infer future outcomes based on an ongoing session or hypothetical situations. Similarly, another approach could merge game sessions that failed to reach the goal in order to detect the most common causes that led to failure.

Finally, our provenance tracking framework is already being used in other studies that include dynamic content balancing, machine learning, deep learning, and multi-agent systems. There is also an ongoing study to use GPU for graph visualization to improve performance when visualizing huge provenance graphs. However, the usage of provenance data can be explored in many other research fields related to games due to its richness and capacity for more abstract analysis. Examples include, but are not limited to, dynamic storytelling generation, automatic content generation and adaptation, content customization based on the player's profile or progress, and improved artificial intelligence by developing characters that react differently based on the player's past events

BIBLIOGRAPHY

AFGAN, E. et al. Galaxy platform for accessible, reproducible and collaborative biomedical analyses: 2016 update. **Nucleic Acids Research**, v. 44, n. W1, p. W3–W10, 2016.

ALTINTAS, I. et al. **Kepler: an extensible system for design and execution of scientific workflows**. 16th International Conference on Scientific and Statistical Database Management, 2004. Proceedings. **Anais**... In: 16TH INTERNATIONAL CONFERENCE ON SCIENTIFIC AND STATISTICAL DATABASE MANAGEMENT, 2004. PROCEEDINGS. jun. 2004

AMBINDER, M. Valve's approach to playtesting: The application of empiricism. **Game Developer Conference (GDC)**, 2009.

ANDERSEN, E. et al. Gameplay analysis through state projection. **Foundations of Digital Games (FDG)**, p. 1–8, 2010.

BAO, Z. et al. PDiffView: Viewing the Difference in Provenance of Workflow Results. **Proc. VLDB Endow.**, v. 2, n. 2, p. 1638–1641, ago. 2009.

BASTIAN, M.; HEYMANN, S.; JACOMY, M. **Gephi: An Open Source Software for Exploring and Manipulating Networks**. Third International AAAI Conference on Weblogs and Social Media. **Anais**...2009Disponível em: <http://www.aaai.org/ocs/index.php/ICWSM/09/paper/view/154>. Acesso em: 29 jan. 2016

BAUCKHAGE, C. et al. **Beyond heatmaps: Spatio-temporal clustering using behavior-based partitioning of game levels**. 2014 IEEE Conference on Computational Intelligence and Games. **Anais**... In: 2014 IEEE CONFERENCE ON COMPUTATIONAL INTELLIGENCE AND GAMES. ago. 2014

BAUCKHAGE, C.; DRACHEN, A.; SIFA, R. Clustering Game Behavior Data. **IEEE Transactions on Computational Intelligence and AI in Games**, v. 7, n. 3, p. 266–278, set. 2015.

BIRANT, D.; KUT, A. ST-DBSCAN: An algorithm for clustering spatial–temporal data. **Data & Knowledge Engineering**, Intelligent Data Mining. v. 60, n. 1, p. 208–221, 1 jan. 2007.

BITON, O. et al. **Querying and Managing Provenance through User Views in Scientific Workflows**. IEEE 24th International Conference on Data Engineering, 2008. ICDE 2008. **Anais**... In: IEEE 24TH INTERNATIONAL CONFERENCE ON DATA ENGINEERING, 2008. ICDE 2008. abr. 2008

BROWN, J. A. et al. **A Machine Learning Tool for Supporting Advanced Knowledge Discovery from Chess Game Data**. 2017 16th IEEE International Conference on Machine Learning and Applications (ICMLA). **Anais**... In: 2017 16TH IEEE INTERNATIONAL CONFERENCE ON MACHINE LEARNING AND APPLICATIONS (ICMLA). dez. 2017

CALLAHAN, S. P. et al. **VisTrails: Visualization Meets Data Management**. Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data. **Anais**...: SIGMOD '06.New York, NY, USA: ACM, 2006Disponível em: <http://doi.acm.org/10.1145/1142473.1142574>. Acesso em: 27 fev. 2014

CHEUNG, K.; HUNTER, J. Provenance Explorer – Customized Provenance Views Using Semantic Inferencing. In: CRUZ, I. et al. (Eds.). . **The Semantic Web - ISWC 2006**. Lecture Notes in Computer Science. [s.l.] Springer Berlin Heidelberg, 2006. p. 215–227.

COX, T.; COX, M. **Multidimensional Scaling, Second Edition**. United Kingdom: CRC Press, 2010.

DAVIDSON, S. B.; FREIRE, J. Provenance and scientific workflows: challenges and opportunities. **ACM's Special Interest Group on Management Of Data (SIGMOD)**, p. 1345–1350, 2008.

DAVISON, A. Automated Capture of Experiment Context for Easier Reproducibility in Computational Research. **Computing in Science & Engineering**, v. 14, n. 4, p. 48–56, 1 jul. 2012.

DEMERS, M. N. **Fundamentals of Geographic Information Systems**. 4 edition ed. Hoboken, NJ: Wiley, 2008.

DEROSA, P. **Tracking Player Feedback To Improve Game Design**. Gamasutra. Disponível em: <http://www.gamasutra.com/view/feature/129969/tracking_player_feedback_to_.php>. Acesso em: 28 jun. 2013.

DIEHL, S. **Software Visualization: Visualizing the Structure, Behaviour, and Evolution of Software**. [s.l.] Springer, 2007.

DIXIT, P.; YOUNGBLOOD, M. Understanding playtest data through visual data mining in interactive 3D environments. **12th International Conference on Computer Games: AI, Animation, Mobile, Interactive Multimedia and Serious Games (CGAMES)**, p. 34–42, 2008.

DRACHEN, A. et al. Guns, swords and data: Clustering of player behavior in computer games in the wild. **Conference on Computational Intelligence and Games (CIG)**, p. 163–170, 2012.

DRACHEN, A. et al. A Comparison of Methods for Player Clustering via Behavioral Telemetry. **Foundations of Digital Games (FDG)**, 2013.

DRACHEN, A. et al. **Guns and guardians: Comparative cluster analysis and behavioral profiling in destiny**. 2016 IEEE Conference on Computational Intelligence and Games (CIG). **Anais**... In: 2016 IEEE CONFERENCE ON COMPUTATIONAL INTELLIGENCE AND GAMES (CIG). set. 2016

DRACHEN, A.; CANOSSA, A. Towards Gameplay Analysis via Gameplay Metrics. **International MindTrek Conference: Everyday Life in the Ubiquitous Era**, MindTrek '09. p. 202–209, 2009a.

DRACHEN, A.; CANOSSA, A. Analyzing spatial user behavior in computer games using geographic information systems. **MindTrek Conference: Everyday Life in the Ubiquitous Era**, p. 182–189, 2009b.

DRACHEN, A.; CANOSSA, A. Evaluating motion: spatial user behaviour in virtual environments. **International Journal of Arts and Technology**, v. 4, n. 3, p. 294, 2011.

DRACHEN, A.; CANOSSA, A.; YANNAKAKIS, G. N. Player modeling using self-organization in Tomb Raider: Underworld. **Computational Intelligence and Games (CIG)**, p. 1–8, set. 2009.

DRACHEN, A.; SCHUBERT, M. Spatial game analytics and visualization. **Computational Intelligence and Games (CIG)**, p. 1–8, ago. 2013.

DUNN, O. J. Estimation of the Medians for Dependent Variables. **The Annals of Mathematical Statistics**, v. 30, n. 1, p. 192–197, mar. 1959.

EBDEN, M. et al. Network Analysis on Provenance Graphs from a Crowdsourcing Application. In: GROTH, P.; FREW, J. (Eds.). . **Provenance and Annotation of Data and Processes**. [s.l.] Springer Berlin Heidelberg, 2012. v. 7525p. 168–182.

EL-JAICK, D.; MATTOSO, M.; LIMA, A. A. B. SGProv: Summarization Mechanism for Multiple Provenance Graphs. **Journal of Information and Data Management**, v. 5, n. 1, p. 16, 13 jul. 2014.

ELLSON, J. et al. Graphviz and Dynagraph — Static and Dynamic Graph Drawing Tools. In: JÜNGER, M.; MUTZEL, P. (Eds.). . **Graph Drawing Software**. Mathematics and Visualization. [s.l.] Springer Berlin Heidelberg, 2004. p. 127–148.

EL-NASR, M.; DRACHEN, A.; CANOSSA, A. (EDS.). **Game Analytics - Maximizing the Value of Player Data**. In: Springer Science & Business Media, 2013.

EL-NASR, M. S.; NGUYEN, T.-H. Glyph: Visualization Tool for Understanding Problem Solving Strategies in Puzzle Games. **Foundations of Digital Games (FDG)**, 2015.

ESMAEILI, H.; WOODS, P. C. **Calm down buddy! it's just a game: Behavioral patterns observed among teamwork MMO participants in WARGAMING's world of tanks**. 2016 22nd International Conference on Virtual System Multimedia (VSMM). **Anais**... In: 2016 22ND INTERNATIONAL CONFERENCE ON VIRTUAL SYSTEM MULTIMEDIA (VSMM). out. 2016

ESTER, M. et al. **A Density-based Algorithm for Discovering Clusters a Density-based Algorithm for Discovering Clusters in Large Spatial Databases with Noise**. Proceedings of the Second International Conference on Knowledge Discovery and Data Mining. **Anais**...: KDD'96.Portland, Oregon: AAAI Press, 1996Disponível em: <http://dl.acm.org/citation.cfm?id=3001460.3001507>

FISHER, R. A. The use of multiple measurements in taxonomic problems. **Annals of human genetics**, v. 7, n. 2, p. 179–188, 1936.

F.R.S, K. P. LIII. On lines and planes of closest fit to systems of points in space. **The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science**, v. 2, n. 11, p. 559–572, 1 nov. 1901.

FULLERTON, T.; SWAIN, C. **Game Design Workshop: A playcentric approach to creating innovative games**. Amsterdam: Morgan Kaufmann/Elsevier, 2008.

GAGNÉ, A. R.; SEIF EL-NASR, M.; SHAW, C. D. Analysis of Telemetry Data from a Real-time Strategy Game: A Case Study. **Computers in Entertainment (CIE)**, v. 10, n. 3, p. 2:1–2:25, dez. 2012.

GIL, Y.; MILES, S. **PROV Model Primer**. Disponível em: <http://www.w3.org/TR/prov-primer/>. Acesso em: 21 mar. 2017.

GOODMAN, L. A. Snowball Sampling. **The Annals of Mathematical Statistics**, v. 32, n. 1, p. 148–170, 1961.

HADIJI, F. et al. Predicting player churn in the wild. **Computational Intelligence and Games (CIG)**, p. 1–8, ago. 2014.

HARRY, M. J.; MOTOROLA UNIVERSITY PRESS. **The Nature of six sigma quality**. Schaumburg, IL: Motorola University Press, 1997.

HIDEN, H. et al. Developing cloud applications using the e-Science Central platform. **Phil. Trans. R. Soc. A**, v. 371, n. 1983, p. 20120085, 28 jan. 2013.

HIGGINS, T. **Unity - 3D Game Engine**. Disponível em: <http://unity3d.com/>. Acesso em: 4 abr. 2018.

HOEKSTRA, R.; GROTH, P. PROV-O-Viz - Understanding the Role of Activities in Provenance. In: LUDÄSCHER, B.; PLALE, B. (Eds.). . **Provenance and Annotation of Data and Processes**. Lecture Notes in Computer Science. [s.l.] Springer International Publishing, 2014. p. 215–220.

HOOBLER, N.; HUMPHREYS, G.; AGRAWALA, M. Visualizing Competitive Behaviors in Multi-User Virtual Environments. **Visualization (VIS)**, p. 163–170, 2004.

HULL, D. et al. Taverna: a tool for building and running workflows of services. **Nucleic Acids Research**, v. 34, n. suppl 2, p. W729–W732, 1 jul. 2006.

HULLETT, K. et al. **Data Analytics for Game Development (NIER Track)**. International Conference on Software Engineering. **Anais**...: ICSE '11.New York, NY, USA: ACM, 2011Disponível em: <http://doi.acm.org/10.1145/1985793.1985952>. Acesso em: 20 fev. 2015

ISBISTER, K.; SCHAFFER, N. **Game Usability: Advancing the Player Experience**. San Francisco, Calif. : Oxford: CRC Press, 2008.

JACOB, L. et al. A Non-intrusive Approach for 2D Platform Game Design Analysis Based on Provenance Data Extracted from Game Streaming. **2014 Brazilian Symposium on Computer Games and Digital Entertainment (SBGAMES)**, p. 41–50, nov. 2014.

JOSLIN, S.; BROWN, R.; DRENNAN, P. The gameplay visualization manifesto: a framework for logging and visualization of online gameplay data. **Comput. Entertain.**, v. 5, n. 3, p. 6, 2007.

JR, J. H. W. Hierarchical Grouping to Optimize an Objective Function. **Journal of the American Statistical Association**, v. 58, n. 301, p. 236–244, 1 mar. 1963.

JUSTESEN, N. et al. Deep Learning for Video Game Playing. **arXiv:1708.07902 [cs]**, 25 ago. 2017.

KANG, J.; LIU, M.; QU, W. Using gameplay data to examine learning behavior patterns in a serious game. **Computers in Human Behavior**, v. 72, p. 757–770, 1 jul. 2017.

KEIM, D. A. Information visualization and visual data mining. **IEEE Transactions on Visualization and Computer Graphics**, v. 8, n. 1, p. 1–8, jan. 2002.

KIM, J. et al. Provenance trails in the Wings/Pegasus system. **Concurrency and Computation: Practice and Experience**, v. 20, n. 5, p. 587–597, 2008a.

KIM, J. H. et al. Tracking real-time user experience (TRUE): a comprehensive instrumentation solution for complex systems. **Human Factors in Computing Systems (CHI)**, p. 443–452, 2008b.

KING, D.; CHEN, S. Metrics for social games. **Presentation at the social games summit, Game developers Conference**, 2009.

KOHWALTER, T. C.; MURTA, L. G. P.; CLUA, E. W. G. Capturing Game Telemetry with Provenance. **Brazilian Symposium on Games and Digital Entertainment (SBGAMES)**, 2017.

KOHWALTER, T.; CLUA, E.; MURTA, L. SDM – An Educational Game for Software Engineering. **Brazilian Symposium on Games and Digital Entertainment (SBGAMES)**, p. 222–231, 2011.

KOHWALTER, T.; CLUA, E.; MURTA, L. Provenance in Games. **Brazilian Symposium on Games and Digital Entertainment (SBGAMES)**, p. 162–171, 2012.

KOHWALTER, T.; CLUA, E.; MURTA, L. Game Flux Analysis with Provenance. **Advances in Computer Entertainment (ACE)**, p. 320–331, 2013.

KOHWALTER, T.; CLUA, E.; MURTA, L. Reinforcing Software Engineering Learning through Provenance. **2014 Brazilian Symposium on Software Engineering (SBES)**, p. 131–140, set. 2014.

KOHWALTER, T.; MURTA, L.; CLUA, E. Filtering irrelevant sequential data out of game session telemetry though similarity collapses. **Future Generation Computer Systems**, v. 84, p. 108–122, 1 jul. 2018.

KOOP, D.; FREIRE, J.; SILVA, C. T. Visual summaries for graph collections. **Visualization Symposium (PacificVis), 2013 IEEE Pacific**, p. 57–64, fev. 2013.

LIU, Y.-E. et al. Feature-based projections for effective playtrace analysis. **Foundations of Digital Games (FDG)**, p. 69–76, 2011.

LOH, C. S.; SHENG, Y. Measuring Expert Performance for Serious Games Analytics: From Data to Insights. In: **Serious Games Analytics**. Advances in Game-Based Learning. [s.l.] Springer, Cham, 2015. p. 101–134.

LOSUP, A. et al. Analyzing Implicit Social Networks in Multiplayer Online Games. **IEEE Internet Computing**, v. 18, n. 3, p. 36–44, maio 2014.

MAHLMANN, T. et al. Predicting player behavior in Tomb Raider: Underworld. **Computational Intelligence and Games (CIG)**, p. 178–185, ago. 2010.

MAHLMANN, T.; DRACHEN, A. Esports Analytics Through Encounter Detection. **Proceedings of the 10th MIT Sloan Sports Analytics Conference**, 2016.

MELLON, L. **Applying metrics driven development to MMO costs and risks**. Redwood City, CA: Versant Corporation, 2009. Disponível em: <http://maggotranch.com/MMO_Metrics.pdf>. Acesso em: 20 fev. 2015.

MILES, S. et al. **Provenance Challenge WIKI**. Disponível em: <http://twiki.ipaw.info/bin/view/Challenge/>. Acesso em: 26 mar. 2013.

MILLER, J. L.; CROWCROFT, J. Avatar movement in World of Warcraft battlegrounds. **Network and Systems Support for Games (NetGames)**, p. 1–6, nov. 2009.

MISSURA, O.; GÄRTNER, T. Player Modeling for Intelligent Difficulty Adjustment. In: **Discovery Science**. Lecture Notes in Computer Science. In: Springer Berlin Heidelberg, 2009. p. 197–211.

MOREAU, L. et al. The Open Provenance Model core specification (v1.1). **Future Generation Computer Systems**, v. 27, n. 6, p. 743–756, 2007.

MOREAU, L.; MISSIER, P. **PROV-DM: The PROV Data Model**. Disponível em: <http://www.w3.org/TR/prov-dm/>. Acesso em: 21 mar. 2013.

MORET, B. Decision Trees and Diagrams. **ACM Computing Surveys (CSUR)**, v. 14, n. 4, p. 593–623, 1982.

MOURA, D.; EL-NASR, M. S.; SHAW, C. D. Visualizing and Understanding Players' Behavior in Video Games: Discovering Patterns and Supporting Aggregation and Comparison. **ACM SIGGRAPH Symposium on Video Games**, Sandbox '11. p. 11–15, 2011.

NASCIMENTO, G. et al. A real-time simulator for ergonomics and displacement evaluations. **Brazilian Symposium on Games and Digital Entertainment (SBGAMES)**, 2010.

NIES, T. D. et al. **Constraints of the PROV Data Model**. Disponível em: <http://www.w3.org/TR/prov-constraints/>. Acesso em: 21 mar. 2013.

NORMAN, G. R.; STREINER, D. L. **Biostatistics: The Bare Essentials**. 3 Pap/Cdr edition ed. Shelton, Conn: People's Medical Publishing House, 2012.

PAO, H.-K.; CHEN, K.-T.; CHANG, H.-C. Game Bot Detection via Avatar Trajectory Analysis. **IEEE Transactions on Computational Intelligence and AI in Games**, v. 2, n. 3, p. 162–175, set. 2010.

PARK, Y. J. et al. A deep learning-based sports player evaluation model based on game statistics and news articles. **Knowledge-Based Systems**, v. 138, p. 15–26, 15 dez. 2017.

PEDERSEN, C.; TOGELIUS, J.; YANNAKAKIS, G. N. Modeling Player Experience for Content Creation. **Transactions on Computational Intelligence and AI in Games (T-CIAIG)**, v. 2, n. 1, p. 54–67, mar. 2010.

PIRKER, J. et al. **How Playstyles Evolve: Progression Analysis and Profiling in <Emphasis Type="Italic">Just Cause 2</Emphasis>**. Entertainment Computing - ICEC 2016. **Anais**...: Lecture Notes in Computer Science. In: INTERNATIONAL CONFERENCE ON ENTERTAINMENT COMPUTING. Springer, Cham, 28 set. 2016Disponível em: <https://link.springer.com/chapter/10.1007/978-3-319-46100-7_8>. Acesso em: 13 abr. 2018

PIRKER, J. et al. Analyzing player networks in Destiny. **Entertainment Computing**, v. 25, p. 71–83, 1 mar. 2018.

PREMIS WORKING GROUP. **Data Dictionary for Preservation Metadata**. OCLC Online Computer Library Center & Research Libraries Group: Implementation Strategies (PREMIS), 2005.

RIJSBERGEN, C. J. V. **Information Retrieval**. 2nd. ed. Newton, MA, USA: Butterworth-Heinemann, 1979.

RIO, N.; SILVA, P. P. Probe-It! Visualization Support for Provenance. In: BEBIS, G. et al. (Eds.). . **Advances in Visual Computing**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007. v. 4842p. 732–741.

ROMERO, R. Successful instrumentation: Tracking attitudes and behaviors to improve games. **Game Developer Conference (GDC)**, 2008.

RUD, O. P. **Business Intelligence Success Factors: Tools for Aligning Your Business in the Global Economy**. 1 edition ed. Hoboken, N.J: Wiley, 2009.

SAAS, A.; GUITART, A.; PERIÁÑEZ, Á. **Discovering playing patterns: Time series clustering of free-to-play game data**. 2016 IEEE Conference on Computational Intelligence and Games (CIG). **Anais**... In: 2016 IEEE CONFERENCE ON COMPUTATIONAL INTELLIGENCE AND GAMES (CIG). set. 2016

SANDER, J. et al. Density-Based Clustering in Spatial Databases: The Algorithm GDBSCAN and Its Applications. **Data Min. Knowl. Discov.**, v. 2, n. 2, p. 169–194, jun. 1998.

SCHOENBLUM, D. **Zero to Millions: Building an XLSP for Gears of War 2**Game developer conference, 2010. Disponível em: <http://www.gdcvault.com/play/1012456>. Acesso em: 6 jun. 2018

SELTZER, M. I.; MACKO, P. Provenance Map Orbiter: Interactive Exploration of Large Provenance Graphs. **TaPP**, 2011.

SHAPIRO, S. S.; WILK, M. B. An Analysis of Variance Test for Normality (Complete Samples). **Biometrika**, v. 52, n. 3/4, p. 591, 1965.

SIBSON, R. SLINK: An optimally efficient algorithm for the single-link cluster method. **The Computer Journal**, v. 16, n. 1, p. 30–34, 1 jan. 1973.

SIFA, R. et al. Predicting Purchase Decisions in Mobile Free-to-Play Games. **AIIDE**, 2015.

SIFA, R. et al. **Predicting Retention in Sandbox Games with Tensor Factorization-based Representation Learning**. 2016 IEEE Conference on Computational Intelligence and Games (CIG). **Anais**... In: 2016 IEEE CONFERENCE ON COMPUTATIONAL INTELLIGENCE AND GAMES (CIG). set. 2016

SIFA, R.; BAUCKHAGE, C.; DRACHEN, A. The Playtime Principle: Large-scale cross-games interest modeling. **Computational Intelligence and Games (CIG)**, p. 1–8, ago. 2014.

THAVASIMANI, P.; CAŁA, J.; MISSIER, P. **Why-Diff: Explaining differences amongst similar workflow runs by exploiting scientific metadata**. 2017 IEEE International Conference on Big Data (Big Data). **Anais**... In: 2017 IEEE INTERNATIONAL CONFERENCE ON BIG DATA (BIG DATA). dez. 2017

THAWONMAS, R.; KASHIFUJI, Y.; CHEN, K.-T. Detection of MMORPG Bots Based on Behavior Analysis. **Advances in Computer Entertainment Technology (ACE)**, ACE '08. p. 91–94, 2008.

THOMPSON, C. Halo 3: How Microsoft Labs Invented a New Science of Play. **Wired Magazine**, v. 15, n. 9, 2007.

THOMPSON, J.; BERBANK-GREEN, B.; CUSWORTH, N. **Game Design: Principles, Practice, and Techniques - The Ultimate Guide for the Aspiring Game Designer**. 1. ed. United States: Wiley, 2007.

WALLNER, G. Play-Graph: A Methodology and Visualization Approach for the Analysis of Gameplay Data. **Foundations of Digital Games (FDG)**, p. 253–260, 2013.

WALLNER, G.; KRIGLSTEIN, S. A spatiotemporal visualization approach for the analysis of gameplay data. **Human Factors in Computing Systems (CHI)**, p. 1115–1124, 2012.

WALLNER, G.; KRIGLSTEIN, S. PLATO: A visual analytics system for gameplay data. **Computers & Graphics**, v. 38, p. 341–356, 1 fev. 2014.

WEBER, B. G. et al. Modeling Player Retention in Madden NFL 11. **Innovative Applications of Artificial Intelligence Conferences (IAAI)**, 2011.

WEBER, B. G.; MATEAS, M. A data mining approach to strategy prediction. **Computational Intelligence and Games (CIG)**, p. 140–147, 2009.

WERNER, C. et al. How Design Style Relates to the Representational Power of Design Outcomes. **NSF-sponsored workshop on Studying Professional Software Design**, 2010.

WILCOXON, F. Individual Comparisons by Ranking Methods. **Biometrics Bulletin**, v. 1, n. 6, p. 80, dez. 1945.

WILLIAMS, D. et al. Looking for Gender: Gender Roles and Behaviors Among Online Gamers. **Journal of Communication**, v. 59, n. 4, p. 700–725, dez. 2009.

WULFF, M.; HANSEN, M.; THURAU, C. **GameAnalytics For Game Developers Know the facts Improve and Monetize**. Disponível em: <http://www.gameanalytics.com/>. Acesso em: 3 dez. 2018.

XIE, H. et al. Predicting Player Disengagement in Online Games. In: CAZENAVE, T.; WINANDS, M. H. M.; BJÖRNSSON, Y. (Eds.). . **Computer Games**. Communications in Computer and Information Science. In: Springer International Publishing, 2014. p. 133–149.

YANNAKAKIS, G. N.; HALLAM, J. Real-time adaptation of augmented-reality games for optimizing player satisfaction. **Computational Intelligence and Games (CIG)**, p. 103–110, dez. 2008.

ZOELLER, G. Development telemetry in video games projects. **Game Developer Conference (GDC)**, 2010.