UNIVERSIDADE FEDERAL FLUMINENSE

SIDNEY ARAUJO MELO

Detecting long-range cause-and-effect relationships in game provenance graphs with graph-based representation learning

> NITERÓI 2019

UNIVERSIDADE FEDERAL FLUMINENSE

SIDNEY ARAUJO MELO

Detecting long-range cause-and-effect relationships in game provenance graphs with graph-based representation learning

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Computação da Universidade Federal Fluminense como requisito parcial para a obtenção do Grau de Mestre em Computação. Área de concentração: Computação Visual

Orientador: Prof. Dr. Esteban Walter Gonzalez Clua

Co-orientador: Profa. Dra. Aline Marins Paes Carvalho

> NITERÓI 2019

Ficha catalográfica automática - SDC/BEE Gerada com informações fornecidas pelo autor



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

SIDNEY ARAUJO MELO

Detecting long-range cause-and-effect relationships in game provenance graphs with graph-based representation learning

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Computação da Universidade Federal Fluminense como requisito parcial para a obtenção do Grau de Mestre em Computação. Área de concentração: Computação Visual

Aprovada em Fevereiro de 2019.

BANCA EXAMINADORA

Prof. Dr. Esteban Walter Gonzalez Clua - Orientador, UFF

fun main fais Convellers

Profa. Dra. Aline Marins Paes Carvalho - Coorientadora, UFF

Prof. Dr. Leonardo Gresta Paulino Murta, UFF

Dr. Troy Costa Kohwalter, UFF

Prof. Dr. José Ricardo da Silva Junior, IFRJ

Niterói 2019

"Sarani mukou e! PLUS ULTRA!!" Kohei Horikoshi

À minha futura esposa, Raquel Azevedo, e aos meus pais, José Ribamar e Angela Zulmira.

Agradecimentos

Primeiramente, aos meus pais, José Ribamar e Angela Zulmira, e à minha irmã Cindy Anne, pelo apoio incomensurável em todas as escolhas, etapas, quedas e jornadas da minha vida. Devo todas as minhas conquistas essencialmente a vocês.

À Raquel Azevedo, minha futura esposa, pelo amor inesgotável e carinho acalentador que me faz enxergar o mundo com um pouco mais de leveza, pela força emprestada nos momentos mais difíceis dessa etapa que se finda, pelo companheirismo e cumplicidade ímpar e, claro, por compartilhar dos melhores hobbies.

Aos meus orientadores Esteban Clua e Aline Paes por terem me guiado habilmente durante a pesquisa que culmina nessa dissertação. Tenho gratidão e orgulho pela confiança depositada em mim, principalmente na reta final dessa etapa. Muito obrigado!

Às famílias Azevedo e Pinheiro por terem me acolhido tão bem, em especial às amadas avós que ganhei: Lourdinha e Zezé.

Ao meu grande amigo Daniel Geovane, pela ajuda sincera e compreensão cirúrgica em todos os momentos.

Aos amigos Lô, Pedro e Yuri pela amizade primordial para meu apreço por essas terras fluminenses. Obrigado por me mostrarem que não se demora em achar pessoas maravilhosas.

Aos amigos da República 49: Nick, Beto, Anderson, Fábio, Beer, Douglas e Heron e Ivan pelos muitos papos, rolês, churrascos, jogatinas, risadas e bagunças.

Aos amigos que fiz no IC, em especial, Ashey, Ícaro, Maira, Felipe, Márcio, Daniel, Paulo, Danilo, Matheus e Léo.

Aos amigos da AMAGames, dos RPGs, de bandas e de outros núcleos que, apesar da distância, continuam fazendo parte da minha vida.

A todos os amigos e colegas que me apoiaram de alguma forma nessa longa caminhada.

Resumo

Game Analytics compreende um conjunto de técnicas para analisar tanto a qualidade de um jogo quanto o comportamento do jogador, a fim de aperfeiçoar a experiência do jogador e/ou dar suporte a decisões de *game design*. Para obter sucesso em tal tarefa, é essencial identificar o que acontece em um jogo (um efeito) e rastrear suas causas. Assim, ferramentas para construir e manipular dados de proveniência de jogos têm sido propostas para capturar relações de causa e efeito durante sessões de jogo para auxiliar o processo de game design, organizando esses dados em grafos de proveniência. No entanto, como a extração de dados de proveniência é guiada por um conjunto de regras pré-definidas estritas estabelecidas pelos desenvolvedores do jogo, a detecção de relações de causa e efeito de longo alcance pode exigir um grande esforço de codificação. Nesse trabalho, apresentamos um framework chamado PingUMiL que aproveita os recentemente propostos *embeddings* de grafos para representar grafos de proveniência em um espaço vetorial latente. Os embeddings aprendidos com os dados são utilizados como atributos de uma tarefa de aprendizado de máquina concebida para detecção de relações de causa e efeito de longo alcance. A avaliação do PingUMiL é realizada por meio de comparações de performance contra métodos clássicos de aprendizado de máquina utilizando as métricas de avaliação precisão, recall e f1. Além disso, a capacidade de generalização do PingU-MiL é testada através de testes de desempenho conduzidos a partir de dados extraídos de grafos de proveniência de jogos de corrida. Os experimentos conduzidos em dois jogos de corrida mostram que (1) o PingUMiL supera a aplicação de métodos clássicos de Aprendizado de Máquina cujos atributos são apenas aqueles obtidos diretamente a partir dos atributos do jogo (ao invés de endereçar a estrutura do grafo), (2) o aprendizado de representações pode ser usado para inferir relações de causa e efeito de longo alcance em grafos de proveniência de jogos não observados e (3) o PingUMiL melhora a detecção de alguns tipos de arestas usando grafos de diferentes jogos do mesmo gênero.

Palavras-chave: Grafo de proveniência, Game Analytics, Aprendizado de representação, Aprendizado de máquina, *Embeddings* de grafos.

Abstract

Game analytics comprises a set of techniques to analyze both the game quality and the player behavior in order to improve player experience and/or support game design decisions. To succeed on that, it is essential to identify what is happening in a game (an effect) and track its causes. Thus, game provenance data extraction and manipulation tools have been proposed to capture cause-and-effect relationships occurring in a gameplay session to assist the game design process, organizing this data as provenance graphs. However, since game provenance data extraction is guided by a set of strict predefined rules established by the game developers, the detection of long-range cause-and-effect relationships may demand huge coding efforts. In this paper, we contribute with a framework named PingUMiL that leverages the recently proposed graph embeddings to represent the provenance graphs in a latent vector space. The embeddings learned from the data pose as the features of a machine learning task tailored towards detecting long-range cause-and-effect relationships. Evaluation of PingUMiL is realized by comparing its performance against classical machine learning methods in terms of precision, recall and f1-score metrics. Also, PingUMiL's generalization capability is tested through performance tests conducted from data extracted of two racing games provenance graphs. The experiments conducted on these two racing games show that (1) PingUMiL outperforms application of classical machine learning methods whose attributes are only the ones directly obtained from game attributes (instead of addressing graph structure), (2) representation learning can be used to infer long-range cause-and-effect relationships in unobserved game data provenance graphs and (3) PingUMiL enhances detection of some edge types using graphs from different games of the same genre.

Keywords: Provenance graph, Game Analytics, Representation learning, Machine learning, Graph embeddings.

List of Figures

1.1	A Venn diagram showing and relating AI technologies. [25]	2
2.1	First step of PinGU implementation: instantiate ${\it Core}$ provenance classes	8
2.2	Second step of PinGU implementation: attach <i>ExtractProvenance</i> provenance classes.	9
2.3	Fourth step of PinGU implementation: create domain-specific provenance tracking functions.	11
2.4	Example of function for adding game-related attributes to a provenance node	12
2.5	Example of function for generating influences.	13
2.6	Fifth step of PinGU implementation: provenance export function attached to an event.	13
2.7	Racing game provenance graph example	14
2.8	Provenance graph elements example	15
2.9	Visual illustration of the GraphSAGE sample and aggregate approach[28]. Source: http://snap.stanford.edu/graphsage	18
4.1	Overview of the proposed framework	26
4.2	Example of a game Y's graph during the preprocessing step	33
4.3	Example of a game Y's graph during inference application step	36
5.1	Racing games screenshots.	39
5.2	$Flying \rightarrow Crash edges examples $	40
5.3	Overall classification results per edge type in CT graphs	48
5.4	Overall classification results per edge type in AC graphs	51
5.5	Overall classification results per edge type in CTAC experiment setting	55

List of Tables

5.1	Total number of CT's positive and negative examples	41
5.2	Total number of AC's positive and negative examples	41
5.3	Total number of edge examples for both CT and AC	42
5.4	Averaged time (seconds) duration of CT's embedding generation step	45
5.5	Averaged time (seconds) duration of CT's classifier training per fold	45
5.6	Averaged results of CT classifiers in a 7-fold cross-validation setting	45
5.7	Averaged performance results per edge for CT	46
5.8	Averaged time (seconds) duration of AC's embedding generation step	48
5.9	Averaged time (seconds) duration of AC's classifier training per fold	48
5.10	Averaged results of AC classifiers in a 4-fold cross-validation setting	49
5.11	Averaged performance results per edge for AC	50
5.12	Averaged time (seconds) duration of CTAC's embedding generation step	51
5.13	Averaged time (seconds) duration of CTAC's classifier training per fold	52
5.14	Mean averaged results of the generalization experiments	52
5.15	Averaged performance results per edge for CTAC	53

List of Abbreviations and Acronyms

AC	:	Arcade Car;
BinG	:	Balance in Games;
CT	:	Car Tutorial;
FN	:	False Negative;
FP	:	False Positive;
GCN	:	Graph Convolutional Networks;
GLA	:	Gaming Learning Analytics;
GNN	:	Graph Neural Networks;
GraphSAGE	:	Graph Sample and Aggregate;
HMM	:	Hidden Markov Models;
LSTM	:	Long Short-Term Memory;
MLP	:	Multi-Layer Perceptron;
MMOG	:	Massively Multiplayer Online Game;
MOBA	:	Multiplayer Online Battle Arena;
NPC	:	Non-Playable Characters;
OPM	:	Open Provenance Model;
PinG	:	Provenance in Games;
PinGU	:	Provenance in Games for Unity;
PPI	:	Protein-protein interaction;
POS	:	Part-of-Speech;
SGD	:	Stochastic Gradient Descent;
SVM	:	Support Vector Machine;
TN	:	True Negative;
ТР	:	True Positive;

Contents

1	Intro	oduction	1
	1.1	Motivation	3
	1.2	Objectives	4
	1.3	Methodology	5
	1.4	Thesis Organization	5
2	The	oretical Background	7
	2.1	Provenance in games	7
	2.2	Graph representation learning	15
		2.2.1 GraphSAGE	17
	2.3	Final considerations	20
3	Rela	ted Work	22
	3.1	Machine Learning in Game Analytics	22
	3.2	Machine Learning in Graphs	24
	3.3	Final considerations	25
4	Ping repr	UMiL: A framework for completing game provenance based on graph-based esentation learning	26
	4.1	Example of a scenario for using PinGUMiL	27
	4.2	Graph capture	28
	4.3	Preprocessing	28
		Graph definition	28

Re	eferen	ces		58
6	Con	clusions		56
	5.3	Threat	s to Validity	54
		5.2.3	Generalization among racing games	51
		5.2.2	Arcade Car	47
		5.2.1	Car Tutorial	45
	5.2	Result	s	43
		5.1.4	Classifier training	42
		5.1.3	Embedding generation	42
		5.1.2	Preprocessing	40
		5.1.1	Graph capture	39
	5.1	Case S	tudy	38
5	Exp	eriment	al Results	38
	4.7	Final o	considerations	37
	4.6	Inferen	nce Application	35
	4.5	Classif	ier training	34
	4.4	Embec	lding generation	33

Chapter 1

Introduction

A game metric is a quantitative measure of one or more attributes of one or more objects that operate in the context of games[18]. Due to the expansion of the game industry and the higher complexity that games have achieved in the past years, game metrics have become a popular, if not essential, tool to support game design and game development. The main idea behind the use of game metrics is to form a basis for analysis using variables and features recorded during game sessions to support decision-making during both the game design and game development[18]. For example, a game metric that measures the mean completion time of a puzzle could be used to determine how hard a puzzle is and where in the game it should appear. However, there is no standard terminology widely accepted to perform in game analytics[18].

Nonetheless, acquiring understandable game metrics is essential to enhance a datadriven game design. This kind of information can be useful for many purposes, such as game balancing[4][69], players behavior understanding[47][66], detection of failures in the game design[76][27], or even enhancing in-game monetizing strategies[21][68].

However, to obtain such metrics, it is necessary to track and remotely gather data from the game sessions, a task known as game telemetry[18]. Thus, game telemetry holds the pillars to intrinsically understand the player's behavior, instead of only relying on feedback that not always retains the true beliefs and motivations of the player [6]. There are different strategies for gathering and storing data collected from games, ranging from raw logs to structured formats such as graphs[72][70]. Particularly, by using a structured, relational representation one can naturally handle objects, entities, characters, their properties and their relationships in a game. With that in mind, provenance graphs, a causality graph that records historic data about objects and their relations, have been proposed for Game Analytics tasks and were recently successfully adapted to record game session history while still denoting the elements of the game and the causal relationships between them [35][38][41][40]. However, there are still few initiatives to discover information and pattern from the data extracted from game provenance, which could bring insights into the process of game design and game development.



Figure 1.1: A Venn diagram showing and relating AI technologies. [25]

Pattern recognition and information discovery are examples of tasks commonly realized by a plethora of Machine learning techniques [53][51][3][14]. However, the performance of machine learning methods is heavily dependent on the choice of data representation (or features) on which they are applied [8]. For this reason, representation learning, i.e., learning representations of the data that make it easier to extract useful information in machine learning tasks, has become a field in itself in the machine learning community[8]. Recent advances in the *Deep Learning* field takes advantage of the internal representation learning performed by deep architectures during whichever task it is trying to solve. As Goodfellow et. al[25] points out, feedforward networks perform representation learning, where the last layer works as a linear classifier and the rest of the network learns to provide a representation to this classifier. Training with a criterion naturally leads to the representation at every hidden layer taking on properties that make the classification task easier[25]. Figure 1.1 shows AI technologies and its relations. Recently, representation learning for structured data (including graphs) has become popular and several techniques have been proposed to generate data representations more suitable for downstream machine learning tasks. In this work, we propose a machine learning framework for provenance graphs based on graph representation learning techniques and evaluate the capabilities of the proposed framework.

1.1 Motivation

Due to its structured representation of game sessions, that is, an annotated causality graph where nodes represent game entities and their respective attributes, and edges represents the causal relationship between these game entities and its attributes, game provenance graphs are a very rich data source. In the context of game provenance, all cause-andeffect relationships in game provenance graphs are materialized as edges. This edges, on their turn, represents the influence between game elements. Existing game provenance solutions^[41] improve the extraction and structured representation of game sessions, subserving further game analysis. This includes, for example, the automatic extraction and representation of direct influence between elements, which are depicted through edges connecting sequential nodes in the provenance graph. Direct influence edges are used to represent cause-and-effect relationships of sequential events or states. Still, there is also a need for domain-specific provenance tracking functions implementation, since different games might have different mechanics. Because of that, these domain-specific functions are implemented by the game developer. One of the needs for domain-specific functions is indirect influences, i.e., a causal relation between non-consecutive events, which represents a long-range cause-and-effect relationship. An example of indirect influence is the influence between jumping and landing. After jumping, it doesn't matter how many events happen before landing, the landing event happens due to the jumping action. However, functions that track indirect influences might grow in complexity if developers need to detect indirect influences, that is, if they need to identify the influence between nodes instantiated along distant timestamps and/or with large path distances. Other difficulties may arise when influences are defined by large sets of rules, conditions or formulas. For example, determining if a player's movement is influenced by a suddenly appearing enemy would have to take into account several variables, such as their positions, orientations, previous and actual movement directions, speed, etc, or assign threat levels regarding all the involved entities attributes. Furthermore, there might still exist important influences not foreseen by the game developer, making difficult their extraction and evidence even in a structured representation such as a provenance graph.

Until now, these difficulties have remained unexplored and no improvements have been proposed regarding indirect influence detection in game provenance graphs. In this work, we propose a framework that uses a recent graph-based representation learning techniques called GraphSAGE [29] in order to detect indirect influence edges and improve game provenance data extraction. Graph representation-learning methods induce a mapping that embeds nodes, edges, or entire (sub)graphs, as points in a low-dimensional vector space[29], which can, therefore, be fed to a downstream machine learning method for tasks such as classification, regression or clustering[3]. By using our framework, the game developer/analyst benefits from automatically detecting edges to complete provenance graphs in at least two scenarios: (i) within a game that lacks some edges, for example, in case of graphs captured with older provenance extraction functions or with older versions of the game, or (ii) the game designer or developer did not realize influences which, in their turn, were not captured during game sessions. Even though in this work we focus on automatically finding indirect influence edges, detection of other types of edges may also be accomplished by the framework with minor modifications.

PingUMiL leverages graph structure and the context in which nodes are inserted in the edge detection task, i.e., given a game object node v, its learned representation will take into account its neighborhood, which comprises v's previous and next (if applicable) feature information, as well as feature information of any other game object with which v is connected to. On the other hand, classical machine learning algorithms are limited to the use of node raw features, that is, only the attribute values attached to the node, ignoring the context in which it is inserted. We assume that the context of a node contains relevant information for influence detection and conduct in order to validate this assumption.

1.2 Objectives

The main objective of this work is to develop a framework for detecting long-range causeand-effect relationships by leveraging methods of representation learning in game provenance graphs. The framework is called PingUMiL. The secondary objectives are:

- 1. to propose a conceptual framework for Machine Learning tasks in game provenance graphs based on representation learning and graph embeddings;
- 2. evaluate if graph representation learning improves the detection of influence edges compared to classical machine learning techniques with raw features;
- 3. evaluate if the framework achieves generalization capability that allows inferring long-range relationships in unobserved but similar provenance graphs.

1.3 Methodology

We develop a conceptual framework for long-range cause-and-effect edges that uses provenance graphs as inputs, employs graph-based representation learning techniques to generate node embeddings for graph nodes and a machine learning library for downstream tasks, in this case, edge detection. Therefore, we investigate both game provenance extraction tools and graph-based representation learning techniques. Thus, we build the conceptual framework containing all steps necessary to perform edge detection, taking into account data transformations required for coupling each framework component into another.

After conceptualizing the framework, we implement PingUMiL around a chosen graphbased representation learning approach and implement scripts that conduct provenance graph input data throughout the framework. In our case, we choose GraphSAGE, a graph representation learning framework that uses neighborhood aggregation architectures for node embedding. We validate our framework with two racing game prototypes. The reasons behinds that are: (1) game provenance data recording was already applied to a racing game prototype in previous researchs, and (2) racing games tend to generate graphs with mostly homogeneous nodes, since game elements (in this case, cars) have mostly the same features.

We evaluate the benefits of our graph-based framework by comparing its performance in terms of quality metrics (precision, recall, f1 score) against feeding classical machine learning methods with raw features extracted from the provenance of two racing games. Also, we take advantage of the similarities between both case studies to investigate the generalization capacity of the proposed framework, by transferring the model learned from one game to another.

1.4 Thesis Organization

The remaining of this work is organized as follows: Chapter 2 presents some background about Provenance in Games and Machine Learning on graphs. Chapter 3 presents related work, discussing recent applications of Machine Learning techniques in the game analytics context and Machine learning techniques for graph-structured problems. Chapter 4 presents our framework proposal. Chapter 5 shows two case studies over racing games, explaining in detail how the framework was applied to them, followed by analysis and discussion of experimental results. Finally, Chapter 6 concludes this work, pointing out contributions, limitations and future works.

Chapter 2

Theoretical Background

This chapter presents two fundamental topics for understanding PingUMiL: Provenance in games and Learning graphs embeddings. In section 2.1, provenance in the game context is defined, followed by a review on previous game provenance works and a discussion on the weak and strong points in the game provenance extraction process. Section 2.2 overviews recent advances in graph representation learning and presents key concepts regarding this topic.

2.1 Provenance in games

Provenance is well understood in the context of art and 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[35]. According to the Open Provenance Model (OPM)[52], provenance of objects is represented by an annotated causality graph, which is a directed acyclic graph enriched with annotation capturing further information pertaining to execution.

The adoption of data provenance in the context of games was first proposed by Kohwalter et al. in the PinG (Provenance in Games) framework[35]. The authors define a mapping between game elements and each type of node of a provenance graph. Summarizing the proposed mapping, one can say that players, enemies and Non-Playable Characters (NPCs) are mapped as *Agent* nodes; items, weapons, potions, static obstacles or any other object used in the game are mapped as *Entity* nodes; and actions and events such as attacking, jumping or interacting with an item are mapped as *Activity* nodes. Causal relationships between game elements are mapped into edges connecting their respective nodes, resulting in a game data provenance graph. The PinG framework was implemented in [39] as a domain-dependant prototype and, later, in [41] as a generic framework for the Unity game engine called Provenance in Games for Unity (PinGU). The PinGU plugin is a domain-independent and low-coupling solution, written in UnityScript (a version of JavaScript used by Unity) that provides easier provenance extraction, requiring minimal coding in the game's existing components [40].

PinGU can be easily integrated into a game by following five steps: instantiate *Core* provenance classes, attach provenance extraction classes, identify actions and interactions, domain-specific provenance tracking function implementation, provenance export[41].

The first step is to instantiate *Core* provenance classes that act as a centralizing server for provenance information[41]. In Unity, that step is completed by creating a game object with two attached components: *ProvenanceController*, responsible for creating both vertices and edges as well as linking them, and *InfluenceController*, responsible for managing the cause-and-effect relationships (influence edges), dealing with possible influences and passing them to the *Provenance Controller* when they materialize in the game[41]. Figure 2.1 shows a Unity game object with the two *Core* provenance classes instantiated.

• Inspector		a -≡
👕 🗹 Provenance		🗌 🗌 Static 🔻
Tag Untagged	‡ Layer Default	\$
🔻 🙏 🛛 Transform		🔯 🌣,
Position	X 65.83183 Y 120.0691	Z -84.18073
Rotation	X 0 Y 0	Z 0
Scale	X 1 Y 1	Z 1
🔻 📘 🗹 Influence Cont	troller (Script)	🔯 🌣,
Script	InfluenceController	0
Provenance	Provenance (Provenance)	ceControlle 🛛
🔻 🌛 🗹 Provenance Co	ontroller (Script)	🔯 🌣,
Script	ProvenanceController	0
	Add Component	

Figure 2.1: First step of PinGU implementation: instantiate Core provenance classes.

The second step is to attach provenance extraction classes to each character or entity in the game and link them to the game object created in the first step[41]. The class *ExtractProvenance* actually creates provenance vertices for the attached game entity and passes it to the *ProvenanceController*. Figure 2.2 shows a Unity game object named Car, the player's avatar in a racing game. For conciseness sake, the Car game object is the only entity in the game that we want to track. Therefore, an *ExtractProvenance* class is attached to the Car game object.

• Inspector	 € +:
😭 🗹 Car	🗌 Static 👻
Tag Untagged	tayer tayer
Prefab Select	Revert Apply
▼ 🙏 Transform	🛐 🗘,
Position	X 55.42717 Y 101.2914 Z -92.72028
Rotation	X 0 Y 130 Z 0
Scale	X 1 Y 1 Z 1
🔻 🙏 Rigidbody	💽 ¢,
Mass	1500
Drag	0
Angular Drag	0
Use Gravity	
Is Kinematic	
Interpolate	None \$
Collision Detection	Discrete +
▶ Constraints	
🔻 📘 🗹 Extract Provenance	(Script) 📓 🎘
Script	ExtractProvenance O
Influence Container	Provenance (InfluenceController) •
Provenance	Provenance (ProvenanceController) ○
🔻 📘 🗹 Car (Script)	
Script	Car O
Suspension Range	0.1
Suspension Damper	50
Suspension Spring Front	18500
Suspension Spring Rear	9250
Brake Lights	CAR_Lights O
Drag Multiplier	X 2 Y 5 Z 1
Throttle	0

Figure 2.2: Second step of PinGU implementation: attach *ExtractProvenance* provenance classes.

The third step is to define relevant actions and interaction among actions. This is a conceptual step closely related to game design and the intended game analytics task. In this step, game developers identify existing classes that contain the actions that they want to track[41]. Let's assume that, in our Car game object example, we want to track actions such as Driving, ChangingGear, Crash, LostControl, Flying (whenever the car is not touching the ground), Landing (when it touches the ground after the Flying action), etc. This is a conceptual step, therefore, the definition of these actions completes the third step for our example.

The fourth step is creating the domain-specific provenance tracking functions and at-

taching it to each entity in the game that has the *ExtractProvenance* module[41]. For each action defined in the third step, a *provenance function* must be implemented. However, these functions follow a four-step recipe:

- Add game-related attributes (e.g., health points, experience points, speed, etc.). Class *ExtractProvenance* provides a function *AddAttribute* to add desired attributes to node attributes.
- Create the appropriate node using functions *NewActivityVertex*, *NewAgentVertex* or *NewEntityVertex*.
- Check for influences. In this step, it is verified whether there is any influence that can affect the current action. This is achieved by using a tag *HasInfluence*, which groups a collection of influences that has something in common, or by an influence ID *HasInfluence_ID*.
- Generate influence. In this step, influences are created using the *GenerateInfluence* function. Also, influences can be created with restrictions such as expiration time leading to the suffix E (i.e., GenerateInfluenceE), the number of times used leading to the C (consuming) suffix (i.e., GenerateInfluenceC) or both (GenerateInfluenceCE). There is also the missable influence, which represents something that should have happened. Functions for missable influences are *GenerateInfluenceM*, *GenerateInfluenceMC*, *GenerateInfluenceMCE*[41].

For conciseness sake, we take as example the Flying action. The Flying action is realized whenever the car loses contact with the ground. This could happen due to high speed values in ramp-like section of the speedway or after a violent crash. In the fourth step, we need to create a domain-specific tracking function for the Flying action that will be called every time the player gets suspended in the air. We name this provenance function *Prov_Flying* and its code can be seen in Figure 2.3.

Note that lines 227-230 follows the recipe for a provenance tracking function. In line 227, game-related attributes are added to current node attributes. The code of the function *Prov_GetPlayerAttributes* can be seen in figure 2.4. First, the function assign values for attributes such as speed, turning rate, velocity, angular velocity, drag vector, etc. Then, PinGU's native function *AddAttribute* is called several times. On each call, two strings are passed: the first one represents the name of the attribute and the second represents the value of the attribute. These two strings are attached to the current node

223	function Prov_Flying()
224	{
225	<pre>if((!isFlying) && (Time.time - oldFlyTime > 2))</pre>
226	{
227	<pre>Prov_GetPlayerAttributes();</pre>
228	<pre>prov.NewActivityVertex("Flying");</pre>
229	<pre>prov.HasInfluence("Player");</pre>
230	<pre>FlyInfluence();</pre>
231	<pre>oldFlyTime = Time.time;</pre>
232	<pre>Debug.Log ("Flying");</pre>
233	<pre>isFlying = true;</pre>
234	}
235	}

Figure 2.3: Fourth step of PinGU implementation: create domain-specific provenance tracking functions.

as an (attribute, value) pair. In line 228, another PinGU native function is called to create the appropriate vertex type. We are creating the Flying Activity Vertex. In line 229, possible influences affecting the current node state of the game object Player are checked. If there is any, an edge is created between existing nodes and the current one.

In line 230, the method *FlyInfluence* is called. This method is shown in Figure 2.5. Three influences are generated by *FlyInfluence*. First, an influence with an expiration time of 2 seconds of type Flying (Crash) for future Crash nodes. That is, an influence edge will be generated if a Crash node is created during the time span of 2 seconds from the creation of the current node. The *ProvInfluence* function for the first line internally calls a *GenerateInfluenceE* function. Second, a consumable influence of type Landing for future Landing nodes. The intuition behind this influence is that if the player is flying, sooner or later, it will land. Whenever that happens, a Landing node will be created and an edge between the last Flying node will connect them. The third influence is very similar to the first one, except it is a LostControl (Crash) influence for future LostControl nodes.

The fifth and final step is to create a function that exports the graphs and use an event as a trigger. Whenever the event occurs, the *ProvenanceController* will save the current graph to an external xml file[41]. Figure 2.6 shows a Unity game object with an *ExportCollider* script. This game object is instantiated in the scene so that every time the player completes a lap, it triggers the provenance export function *Prov_Export*, which, in its turn, saves the provenance graph to an external xml file.

```
function Prov GetPlayerAttributes()
Ł
        currentSpeed = rigidbody.velocity.magnitude;
        currentTurn = EvaluateSpeedToTurn(currentSpeed);
        deltaTime = Time.time - oldTime;
        deltaSpeed = currentSpeed - oldSpeed;
        velocityVector = rigidbody.velocity;
        angularVelocity = rigidbody.angularVelocity;
        velocityVector.Normalize();
        angularVelocity.Normalize();
        dragVector.Normalize();
       prov.AddAttribute("Throttle", throttle.ToString());
prov.AddAttribute("Speed", currentSpeed.ToString());
prov.AddAttribute("CurrentGear", currentGear.ToString());
prov.AddAttribute("CurrentEnginePower", currentEnginePower.ToString());
prov.AddAttribute("TurnRate", currentTurn.ToString());
prov.AddAttribute("CarMass", rigidbody.mass.ToString());
prov.AddAttribute("DeltaTime", deltaTime.ToString());
prov.AddAttribute("VelocityVector_X", velocityVector.x.ToString());
prov.AddAttribute("VelocityVector_Y", velocityVector.y.ToString());
prov.AddAttribute("VelocityVector_Z", velocityVector.z.ToString());
prov.AddAttribute("AngularVelocity_X", angularVelocity.x.ToString());
prov.AddAttribute("AngularVelocity_Z", angularVelocity.z.ToString());
prov.AddAttribute("AngularVelocity_Z", angularVelocity.z.ToString());
prov.AddAttribute("IngularVelocity_Z", angularVelocity.z.ToString());
        prov.AddAttribute("Throttle", throttle.ToString());
        prov.AddAttribute("DragVector_X", dragVector.x.ToString());
        prov.AddAttribute("DragVector_Y", dragVector.y.ToString());
        prov.AddAttribute("DragVector_Z", dragVector.z.ToString());
        Influences();
        oldSpeed = currentSpeed;
        oldTime =
                            Time.time;
        deltaTurn = currentTurn
                                                               oldTurn;
        oldTurn = currentTurn;
```

Figure 2.4: Example of function for adding game-related attributes to a provenance node.

Once these five steps are completed, every gameplay session of the game will be recorded according to all the implemented tracking functions and exported as an xml file containing its respective provenance graph. After a provenance graph is exported, it could be visualized in the Prov Viewer tool[37]. Prov Viewer is a graph based visualization tools which has been developed for post-game session analysis.

One of the most prominent advantages of provenance graphs is its richness of detail, i.e. the data collected at fine-grain. The Figure 2.7 presents an example of a racing game provenance graph and its components. The graph is plotted using the player node's coordinates X and Z within the game space so that it is possible to have a general view of the race course and, for that specific scenario, follow the player trajectory and other events by traversing the graph. It is important to notice that in the scenario presented in

function FlyInfluence()	
{	
<pre>ProvInfluence("Crash", "Flying (Crash)", countInf.ToString(), 0, Time.time + 2); ProvInfluence("Landing", "Landing", countInf.ToString(), 0);</pre>	
ProvInfluence("LostControl", "LostControl (Crash)", countInf.ToString(), 0, Time.time + 2	2);
}	

Figure 2.5: Example of function for generating influences.

0 Inspector						a .	=
👕 🗹 ExportProv						Static	Ŧ
Tag Untagged		‡ Layer		Default			ŧ
🔻 🙏 Transform						2	≱,
Position	Х	36.27403	Y	103.028	Z	-79.53888	
Rotation	Х	0] Y	0	Z	0	
Scale	Х	1	Y	1	Z	1	
⊳ 🤪 🗹 Box Collider						[2] \$	\$₽,
🔻 🌛 🗹 Export Collider (Scrip	t)					i 🛛 🕯	≵,
Script		ExportCollide	er				0
Xml Name	C	ar_Tutorial					
Count Laps]					
Lap	0						
	A	dd Compone	nt				

Figure 2.6: Fifth step of PinGU implementation: provenance export function attached to an event.

Figure 2.7, a single lap has generated 140 nodes and more than 700 edges in less than a two-minute gameplay session.

In Figure 2.8(a), it is possible to observe all the attributes of a car controlled by the player during the *Driving* activity. While PinGU implements several methods to facilitate provenance capture, game developers must write domain-specific provenance tracking functions and attaching it to each entity in the game [40]. Therefore, the amount of data gathered in a single node depends on the developer's design and his analytic choices.

In summary, the implementation of the data extraction algorithms and the events happening within a game session directly influence the amount of generated data[40]. In addition, this rich and raw provenance data can also be used for machine learning tasks, to describe hidden patterns, aid the game maintenance and help in future developments. These are the main assumptions of this research.

Figure 2.8(b) presents a sequence of activity nodes connected by edges which represent the causal relationship between these activities. For example, the presented sequence could be interpreted as "the player crashed due to its high speed while using the hand brake" by observing the sequence of *ChangedGear*, *HandBrake*, *ReleasedHandBrake* and *Crash* activity nodes and their *speed* attribute values, which quickly increases between



Figure 2.7: Racing game provenance graph example.

the last two nodes of the sequence.

Edges, in their turn, are extracted according to its influence range. Direct influence edges, i.e. edges connecting nodes sequentially are automatically extracted by PinGU. Indirect influence edges, similarly to node attributes extraction, are extracted through domain-specific provenance tracking functions. These indirect influences are provided by the game designer which, in turn, guides these functions implementation.

The amount of data gathered during a game session to be included in a game provenance graph implies in huge time and manual efforts to the game analysis process. Moreover, much effort has also to be put by the game developer and the game designer when implementing or adapting domain-specific provenance extraction algorithms, besides, the possibility of not preconceiving all the types of influences. Thus, we leverage recent advances of machine learning techniques acting on graph-structured data to suggest their use to detect long-range influences as a first attempt to enhance and automate game



Figure 2.8: Provenance graph elements example.

provenance data extraction.

2.2 Graph representation learning

Graphs are a fundamental data structure in computer science[19]. Real-world data commonly containing relationships (hierarchy, interactions, geometry) between entities are often represented as graphs: social networks[73], molecular structures[46], web data[1], *etc.* In these domains, each object is represented as a node, and the relation between these objects are modeled as edges.

Recently, efforts to employ graph-structured data in machine learning tasks have gained attention[61][56][55][26][12][28]. Most machine learning applications seek to make predictions, discover new patterns or classify nodes by transforming graph-structured data into feature information [43][67][29].

Many graph representation learning methods have been proposed in order to incorporate information about the structure of the graph[56][71][33][28]. The idea behind these representation learning approaches is to learn a mapping that embeds nodes, or entire (sub)graphs, as points in a low-dimensional vector space, \mathbb{R}^d . The goal is to optimize this mapping so that geometric relationships in this learned space reflect the structure of the original graph[29]. Vector representations resulting from mapping nodes into the learned space are called node embeddings. These node embeddings, on their turn, can be fed to downstream machine learning tasks such as classification, clustering, etc.

Hamilton et al. also points out that various node embedding methods rely on the four components below[29]:

- A pairwise proximity function, which measures how closely two connected nodes are.
- An encoder function, which generates node embeddings.
- A decoder function, which reconstructs pairwise proximity values from the generated embeddings.
- A loss function, which determines the quality of the pairwise reconstructions in order to train the model.

Earliest methods such as the Laplacian Eigenmap [7], HOPE [55], DeepWalk [56], node2vec [26] generate vector representations for each node independently. These methods, which can be categorized as *direct encoding* methods [29], rely on matrix factorization approaches for dimensionality reduction [7] (Laplacian Eigenmaps, HOPE) or on random walk statistics (DeepWalk, node2vec). However, these methods do not consider node attributes during the encoding. This is a major drawback since node attributes can be highly relevant to a node's representation.

Thus, convolutional approaches have been proposed to solve this problem. In general, these approaches generate a node embedding iteratively. At the first step, the node embedding is initialized with the node's features. At each iteration, the nodes aggregate their neighbors' embeddings, generating new embeddings. These approaches determine the node embedding according to its surrounding neighborhood attributes. Therefore, these methods are also called neighborhood aggregation methods. Examples of these methods are Graph Convolutional Networks (GCN) [33], Column Networks [57] and GraphSAGE [28]. Still, these methods have been proposed for homogeneous, non-layered graphs and graphs in which proximity is more relevant to embeddings than the structural role of the nodes. Very recently, extensions and solutions have been proposed for handling these limitations[17][75][58].

According to the complexity of the domain, a graph might contain several types of nodes and edges and also data pertinent to each of its components. Therefore, graphs not only capture structural data but also allow this structured data to be very rich in features. In a number of real-world problems, such as the ones handling drug structures, graphs may have multiple types of nodes, multiples types of edges and features attached to each type of node or edge. This is also the case of the game provenance graphs we tackle here.

Chang et al. present a solution for graphs with different types of nodes by using different encoders for each type of node and pairwise decoders for each pair of node types[13]. This approach is also employed by Schlichtkrull et al[?].

Another relevant topic to the representation learning is the Graph Neural Networks (GNN). Closely related to the neighborhood aggregation methods, feature aggregation and subgraph embedding are achieved through recurrent neural network layers built over the graph structure. The method proposed by Scarselli et al[61] relies on the resultant recurrent neural network with a set of parameters that ensures convergence. Once the embeddings have converged, they are aggregated for the entire (sub)graph and this aggregated embedding is used for subgraph classification[29].

Li et al.[43] proposes extensions to the GNN by implementing Gated Recurrent Units[15] and backpropagation through time to improve the GNN. The Gated Neural Network proposed by Li is capable of outputting intermediate embeddings of subgraphs. Velickovic et al.[67] presents the Graph Attention Networks, which are based on attention mechanisms to compute representations. Attention mechanisms allow for dealing with variably sized inputs, focusing on the most relevant parts of the input to make decisions. In the graph context, the attention operations are employed only in the neighborhood of a node.

2.2.1 GraphSAGE

The GraphSAGE (Graph Sample and Aggregate) framework[28], as previously mentioned, is a convolutional approach for graph representation learning that uses neighborhood aggregation methods to build up node embeddings. In this subsection, we describe in details GraphSAGE and its features and justify its choice as graph embedding tool for this work. We start by explaining its embedding generation process, followed by its parameter learning function. Finally, we describe GraphSAGE's aggregator architectures.



Figure 2.9: Visual illustration of the GraphSAGE sample and aggregate approach[28]. Source: http://snap.stanford.edu/graphsage

Figure 2.9 illustrates the approach implemented by GraphSAGE. Its main goal is to learn useful representations by aggregating features from a node's local neighborhood iteratively and then use graph-based loss functions to fine-tune weight matrices and aggregation functions' parameters. This graph-based loss function enforce similarities on representations of nearby nodes.

Algorithm 1: GraphSAGE embedding generation algorithm		
input : Graph $G(V, E)$; input features $\{\mathbf{x}_v, \forall v \in V\}$; depth K; weight matrices		
$\mathbf{W}^k, \forall k \in \{1,, K\};$ non-linearity σ ; differentiable aggregator functions		
$AGGREGATE_k, \forall k \in \{1,, K\};$ neighborhood function $N: v \to 2^V$		
output: Vector representations $\mathbf{z}_v, \forall v \in V$		
$1 \ \mathbf{h}_v^0 \leftarrow \mathbf{x}_v, \forall v \in V;$		
2 for $k = 1K$ do		
3 for $v \in V$ do		
$4 \left \mathbf{h}_{N(v)}^{k} \leftarrow AGGREGATE_{k}\left(\left\{\mathbf{h}_{u}^{k-1}, \forall u \in N(v)\right\}\right);$		
5 $\mathbf{h}_{v}^{k} \leftarrow \sigma \left(\mathbf{W}^{k} \cdot CONCAT \left(\mathbf{h}_{v}^{k-1}, \mathbf{h}_{N(v)}^{k} \right) \right) ;$		
$6 \left[\mathbf{h}_{v}^{k} \leftarrow \mathbf{h}_{v}^{k} / \left\ \mathbf{h}_{v}^{k} \right\ _{2}, \forall v \in V ; \right]$		
7 $\mathbf{z}_v \leftarrow \mathbf{h}_v^K, \forall v \in V$		

Algorithm 1 describes the embedding generation process. GraphSAGE's embedding generation approach employs K aggregator functions (denoted as $AGGREGATE_k, \forall k \in$ $\{1, ..., K\}$ in order to aggregate information from node neighbors and a set of weight matrices $\mathbf{W}^k, \forall k \in \{1, ..., K\}$ to propagate information between different layers of the model. For the sake of conciseness, let's assume that parameters for both aggregator functions and weight matrices are already learned. Line 1 initializes the algorithm by populating vectors \mathbf{h}_v^0 with input features \mathbf{x}_v . Lines 2-6 presents the outer loop with K steps that generate a node representation \mathbf{h}^k for each step. First, each node $v \in$ V aggregates representations of the nodes in its immediate neighborhood sampled by function N, $\{\mathbf{h}_u^{k-1}, \forall u \in N(v)\}$ into a single vector $\mathbf{h}_{N(v)}^k$ (Line 4). This aggregation step depends on the previous iteration of the outer loop, defined in \mathbf{h}_u^{k-1} , or on the case base k = 0, defined as the input node features in Line 1. After aggregating the neighborhood of a node, GraphSAGE concatenates node's current representation \mathbf{h}_v^{k-1} with the aggregated neighborhood vector $\mathbf{h}_{N(v)}^k$. The concatenated vector is fed through a fully-connected layer with nonlinear activation function σ , which outputs representations \mathbf{h}_v^k to be used at the next step of the algorithm. Final representation, obtained after K steps, is output as $\mathbf{z}_v[28]$.

In order to learn useful, predictive representation in an unsupervised setting, Graph-SAGE applies a graph-based loss function to the output representations $\mathbf{z}_u, \forall u \in V$ and tunes the weight matrices $\mathbf{W}^k, \forall k \in \{1, ..., K\}$ and parameters of the aggregator functions via stochastic gradient descent[28]:

$$J_G(\mathbf{z}_u) = -\log\left(\sigma\left(\mathbf{z}_u^{\top} \mathbf{z}_v\right)\right) - Q \cdot \mathbb{E}_{v_n \sim P_n(v)} \log\left(\sigma\left(\mathbf{z}_u^{\top} \mathbf{z}_{v_n}\right)\right)$$
(2.1)

In equation 2.1, v is a node that co-occurs near u n fixed-length random walk, σ is the sigmoid function, P_n is a negative sampling distribution and Q defines the number of negative samples. This function encourages nearby nodes to have similar representations, while enforcing disparate nodes have different representations[28].

GraphSAGE presents three base aggregator functions: Mean aggregator, LSTM aggregator and Pooling aggregator.

The Mean aggregator takes the elementwise mean of the neighbors' representations vectors $\mathbf{h}_{u}^{k-1}, \forall u \in N(v)$. A GCN aggregator is derived by replacing lines 4 and 5 in algorithm 1 with the following[28]:

$$\mathbf{h}_{v}^{k} \leftarrow \sigma \left(\mathbf{W} \cdot MEAN \left(\left\{ \mathbf{h}_{v}^{k-1} \right\} \cup \left\{ \mathbf{h}_{u}^{k-1}, \forall u \in N(v) \right\} \right) \right)$$
(2.2)

The LSTM aggregator is based on an LSTM architecture[30]. Since LSTMs process their inputs in a sequential manner, GraphSAGE's LSTM aggregator applies LSTMs to a random permutation of the node's neighbors[28]. In Pooling aggregators, each neighbor's vector is independently fed through a fullyconnected neural network, followed by an element-wise pooling operation to aggregate information across neighbor set[28]. The equation 2.3 shows a max-pooling aggregator, where max denotes the element-wise max operator and σ is a nonlinear activation function. GraphSAGE provides both max and mean-pooling aggregators.

$$AGGREGATE_{k}^{pool} = max\left(\left\{\sigma\left(\mathbf{W}_{pool}\mathbf{h}_{u_{i}}^{k} + \mathbf{b}\right), \forall u_{i} \in N(v)\right\}\right)$$
(2.3)

In this research, we opted for GraphSAGE (Graph Sample and Aggregate) framework due to the following reasons:

- GraphSAGE is an inductive approach to node embedding generation, which facilitates generalization across graphs with the same form of features.
- It comprises an unsupervised setting, which emulates situations where node features are provided to downstream machine learning applications, as a service or in a static repository.
- It has multiple aggregator architectures, i.e., functions defined for aggregating node embeddings according to its sampled neighborhood.

It is also worth mentioning that GraphSAGE achieved state-of-the-art performance on node classification tasks on the Web of Science citation and a Reddit posts datasets[28]. Also, the performance of unsupervised GraphSAGE is reasonably competitive with the fully supervised version, indicating that GraphSAGE can achieve strong performance without task-specific fine-tuning[28].

2.3 Final considerations

In this chapter, we defined provenance in games, presented how provenance graphs were conceptualized and implemented in the context of games and discussed relevant characteristics on both provenance graphs, specially its huge amount of detailed data and the difficulty on implementing indirect edges extraction algorithms. Then, we introduced graph representation learning and reviewed several approachs and techniques that leverage graph structure and permit graphs to be fed to downstream machine learning tasks. We also presented the GraphSAGE framework, the convolutional graph representation learning tool employed in PingUMiL. In the next chapter, we contextualize the PingU-MiL in the context of machine learning applied to game analytics and machine learning applied on graph structured data.

Chapter 3

Related Work

This chapter presents previous works related to our in two contexts. First, some works relating Machine Learning and Game Analytics are presented in section 3.1 in order to contextualize PingUMiL among recent Game Analytics research. Following that, sections 3.2 presents works relating Machine Learning and graph-structured problems, in order to show what kind of tasks and challenges have been attacked and solved by graph-based machine learning solutions.

3.1 Machine Learning in Game Analytics

Machine Learning in Game Analytics is a recent research area, whose advent is related to technological advances in cloud computing, machine learning, infrastructures, etc. Recent works have explored tasks such as procedural generation of content, prediction (recommendation systems, event prediction)[63] and game learning analytics[62][10].

Block et al.[10] presents a procedurally generated content tool called *Echo* which uses live and historic match data to detect extraordinary player performances and dynamically translates interesting data points into audience-facing graphics. *Echo* compares the performance metrics of each player to thousand of historic, calculating how many percents of historic performances are exceeded in the game session.

Schubert et al.[62] present a technique for segmenting matches of the Multiplayer Online Battle Arena (MOBA) game DOTA into spatio-temporally defined components called encounters. Encounters happen when two or more units (avatars controlled by players) from opposing teams are in range to affect each other. Encounter dynamics are modeled as graphs where each node represent a unit and edges connects closely positioned
units. Then, encounters are defined as a sequence of situations in time and space involving the same units. Metrics such as unit values, defeated units and resources earned per team are captured during these encounters. Using an algorithm to extract encounters while parsing through replay files and encounter-based analysis, it was possible to break down complex dynamics into manageable components and train a logistic regression classifier to predict the outcome of an encounter based on its initial conditions.

In [63], Tamassia et al. investigate player retention prediction in the Massively Multiplayer Online Game (MMOG) Destiny using Hidden Markov Models (HMM). Sequences of weeks are labeled according to the total playtime of a player and features such as mean lifespan, kill/death ratio and absence are used to predict if a player is about to churn, i.e. leave the game indefinitely and discontinue to be a costumer.

Freire et al.[24] combines the educational goals of Learning Analytics and tools and technologies from Game Analytics into Gaming Learning Analytics (GLA) and defines a conceptual model of the tasks required to analyze players interactions in serious games. The main objective of GLA is to improve the practical applicability of serious games. Recently, Kickmeier[32] combined learning performance metrics and log files from serious games to predict learning performances in serious games.

Regarding game provenance data, most recent works have combined provenance with data mining and logical programming. In [36], Kohwalter et al. proposed similarity collapse algorithms based on the DBSCAN algorithm[20], a classic clustering algorithm, in order to filter sequential information that has similar values or represents the same states inside a provenance graph. In [22], Figueira et al. present BinG (Balance in Games), a framework to collect and process game provenance information to produce parameters for game dynamic balancing. BinG transforms gathered data into facts and rules in a logical programming paradigm, which are used by a Balancing Model in order to improve game experience.

The aforementioned works use game analytics to comprehend events occurring during a game session and use machine learning for tasks such as prediction and performance improvement related to the game session itself. On the other hand, the present dissertation is an attempt to support and improve a game analytics process (in this case, Provenance in Games[35]) regarding causal relationships between game components using machine learning. To the best of our knowledge, there are no works directly related to the framework proposed in this paper. Also, PingUMiL could support previously mentioned examples by providing node embeddings as alternative inputs that take into account both node features and the context in which the node is inserted.

3.2 Machine Learning in Graphs

The most common machine learning tasks in the graph domain are clustering, node classification, and link (edge) prediction. Most recent solutions have proposed the use of node embeddings for these tasks. Also, applications related to graphs and machine learning belongs to a wide range of domain, such as computational social science and computational biology[29].

One example of clustering for graphs is aforementioned similarity collapse clustering in [36]. In [23] several applications are presented for clustering tasks in domains such as biological and social networks. For example, in [59], a hierarchical clustering technique is employed to detect clusters of proteins involved in the process leading microorganisms to a filamentous form. In [56], DeepWalk is used on Zachary's Karate network[73] to generate node embeddings which, in their turn, are used to detect communities. Rosvall and Bergstrom used a clustering technique based on compressing random walks information of a citation graph of over 6000 scientific journals to derive a map of science[60].

Node classification is perhaps the most common task for graph-structured problems. In most cases, a node classification task is a form of semi-supervised learning, where labels are only available for a small proportion of nodes, with the goal being to label the full graph based only on this small initial seed set[29]. In [26], node embeddings are employed on multi-label classification tasks in the following datasets: Blog Catalog[74], a network of social relationships of bloggers where labels represent blogger interests; Protein-Protein Interactions (PPI)[11], a PPI network in which nodes contained protein information and labels are obtained from hallmark gene sets[44] and represent biological states; and Wikipedia[48], a co-occurrence network of words appearing in the first million bytes of the Wikipedia dumb, where labels represent Part-of-Speech (POS) tags[65]. Similar tasks are also presented in [33][56][64][67].

Link prediction is the task whose goal is to predict missing edges or edges that are likely to form in the future[5]. In [9], Berg et al. propose a graph auto-encoder framework based on differentiable message passing on the bipartite interaction graph, a user-item graph relating users and movie ratings. Applications of this task share several domains with node classification task. For example, Grover et al.[26] also applies link predictions task in the PPI and Facebook datasets. Similarly, [2] also realizes link prediction in the PPI dataset. Link prediction is also used in recommender systems. For example, in [42], a graph kernel-based recommendation framework is proposed for link prediction in a useritem interaction graph. Another example is presented in [45], where Liu and Kou cast the "Who Rated What" task as a link prediction problem. The goal of the task is to predict whether users will review movies in the future. The proposed technique was applied in a huge Netflix user-item interaction dataset.

The framework presented in this dissertation relies on node embedding generation techniques, that is, nodes are mapped to a latent vector space. Once embedding generation is realized, it is possible to use node embeddings for all the previously mentioned tasks. The PingUMiL experiments described in the present dissertation are similar to the aforementioned link prediction examples. Yet, one could use PingUMiL generated node embeddings for both node classification or clustering.

3.3 Final considerations

In this chapter, we reviewed work relating Machine Learning and Game Analytics, and Machine Learning and graph-structured problems. Previous works related to game analytics use Machine Learning to improve understading of the game session itself, while we attempt to support and improve the game analytics process and its products (in this case, provenance graphs). Works related to graph-structured problems presents the most frequent Machine Learning tasks for this data structure, including link prediction.

In the next chapter, we introduce, explain in details and discuss the PingUMiL framework, its concepts, steps and components.

Chapter 4

PingUMiL: A framework for completing game provenance based on graph-based representation learning

In this chapter we propose a framework for detecting long-range cause-and-effect relationship by leveraging methods of representation learning in game provenance graphs. Also, we intend to define a conceptual framework that could be easily adapted for other Machine Learning tasks. By the end of this chapter, all steps and tools employed by the framework should be explained in details.



Figure 4.1: Overview of the proposed framework.

The general procedure of the framework proposed in this work, named as PingUMil¹ is to induce a latent representation of the edges in a provenance graph so that they become the examples in a classification task, aiming at inducing a model that discriminates whether an edge is of a specific type or not. For example, we would like to know whether or not a given edge example is a long-range influence edge or not.

An overview of PingUMil is shown in Figure 4.1. A set of provenance graphs is the input to the whole framework. These graphs must be preprocessed due to node hetero-

 $^{^{1}} https://github.\ com/sidneyaraujomelo/PingUMiL$

geneity and the definition of balanced sets of positive and negative example edges for the downstream classification tasks. After the preprocessing, graphs are fed to an embedding generation technique which outputs node embeddings. Then, edges from positive and negative example edge sets are encoded using their connecting nodes' embeddings. Encoded edges are finally fed to a classifier training algorithm. The resulting classifier should be able to detect edges similar to the examples edge sets and generalize this detection capability to semantically analogous edges in graphs not seen in the training phase.

In section 4.1, we define a guiding example for better explaining our framework. Then, we divided the proposed framework for detecting edges in four steps, discussed in their respective sections: graph capture (4.2), pre-processing (4.3), embedding generation (4.4), classifier training (4.5). Finally, the use of the resulting model is discussed in the section 4.6. Before detailing the steps, we introduce a guiding example that is revisited in each of the following sections.

4.1 Example of a scenario for using PinGUMiL

In this section, we present an intentionally simplified scenario to illustrate PingUMiL usage. Let us consider a game development studio X which employs Provenance in Games for Game Analytics features in their game racing game Y. Game studio X stores every played game session as a provenance graph. Some time after releasing the game, which means that X has numerous provenance graphs stored, Bob, a game analytics specialist from the company X, notices an unforeseen influence between non-consecutive player actions while watching gameplay videos: most players tended to start drifting whenever they perceived an "avoidable collision" instead of breaking.

After an in-depth analysis of the graphs, Bob finds no trivial set of rules that determine this new influence, i.e., the player perception of what defines an "avoidable collision", even though he can detect it by observing the context in which the influence is realized. Bob could then start to manually insert new influence edges for all game Y graphs. Another alternative would be to use a method that, given examples of edges on provenance graphs, could find the hidden pattern underlying the influence cases observed by Bob. We present in the following sections a method that aims solving this and similar problems regarding edge detection.

4.2 Graph capture

The first step of PingUMiL is to capture provenance graphs from the played sessions of a game. This step can be skipped if there are already captured provenance graphs. If graph capture is needed, provenance extraction algorithms must be implemented in the game of interest, using tools such as the PinGU plugin[41]. Then, game sessions must be played and recorded in order to generate a dataset of provenance graphs.

Regarding the posterior machine learning tasks, it is also necessary to provide positive and negative examples for training the classifier. Thus, developers should consider coding some long-range influence edges extraction functions or identify these manually in generated provenance graphs. These edges will be referred to as positive example edges from now on.

In our scenario, graph capture is already realized by game studio X as part of their game analytics process.

4.3 Preprocessing

The preprocessing step is fundamental for both the node/edges embedding generation and the training of the classifier tasks. In the case of provenance graphs containing longrange influence edges, these edges must be removed. This is necessary for embedding generation because the learned node representations must simulate a graph without longrange indirect edges. Generating node embeddings containing the long-range indirect edges would encode the same relationships we want to detect and introduce a bias to the classification task. These removed edges are going to compose the positive example edges set. In its turn, this set of edges and their connecting node's attributes are used to define sample criteria for sampling negative example edges. This step is explained in details as follows.

Graph definition Consider a graph $G = (V, E, T_v)$. A node v is defined as $v \in V = (x_v, t_v)$ where $x_v \in \mathbb{R}^{n(\tau(v))}$ is the node feature vector, τ is a mapping function that maps node into a type $t \in T_v$ and n is a function that maps a type of node t into an integer that represents the dimension of the type t. $T_v = \{t_i \in \mathbb{R}^{d_i}, i = 1, ..., |T_v|\}$ is the set of node types, where d_i is the number of dimensions of the type t_i . Every dimension of t_i represents a label $l \in L$ for node attributes such as "speed", "hp", "damage", etc. In

summary, nodes have a vector or features and its dimension depends on the type of the node. Every type t of a node defines a set of labels so that each label corresponds to a value in the node feature vector. Using this notation, an activity node with attributes HP = 10 and Speed = 5 and provenance label *Running* could have, for example, a type $t_x = (provenance_type, provenance_label, hp, speed)$ and $x_v = (activity, running, 10, 5)$. Edges are defined as $(v, v') \in V \times V$. Even though edge types are present in provenance graphs, they are not inserted in this notation for conciseness sake.

Embedding generation is realized through GraphSAGE [28], which takes as input graphs with homogeneous nodes, i.e., nodes with the same set of features. Provenance graphs with heterogeneous nodes must, therefore, be mapped into homogeneous nodes. That can be achieved by creating a new type $t' = \{l \mid l \in t_i, i = 1, ..., |T_v|\}$. A homogeneous node is defined as $v' = (x_{v'}, t')$, where $x_{v'} \in \mathbb{R}^{|t'|}$ is composed of the original values of node feature vector x_v , respecting labeling order and default values for previously unaddressed features. Once all the nodes are homogeneous, any non-numeric attribute must be mapped into one-hot-vector representations, i.e., a k-dimensional binary vector with a single '1' value, where k is the number of possible values of the non-numeric attribute and the position of the '1' value represents the attribute value.

As mentioned in section 4.2, positive and negative example edges are necessary for posterior binary classification tasks. Positive examples might be obtained through:

- (a) provenance extraction algorithms, in case of easy or relaxed rules for long-range influence edges,
- (b) through an expert's analysis and identification.

In the first case, captured long-range influences must be removed from the graph. This step is necessary to make the classifier able to detect long-range edges from the node embeddings generated using only direct edges. In other words, if the long-range edges are already part of the graph when the node embeddings are generated, the classifier would assume from the embeddings that there is nothing left to be linked, as the desired edges to be induced would be already there. From now on, we refer as $G^- \subset G$ and $E^+ \subset E$ the resulting graph without the long-range edges and the set of positive example edges, which are exactly the long-range edges, respectively.

Positive example edges are used to determine the sample criteria for negative example edges in G^- . Let L_v be the set of node's provenance label values, we define edge labels as $L_e = l_i \rightarrow l_j$ where $l_i, l_j \in L_v$. Thus, we define L_{E^+} as the set of labels from positive example edges. This defines the first criteria for acquiring negative example edges, i.e., a negative example edge e_i^- must have an edge label $l_k \in L_{E^+}$. Therefore, the set of negative example edges labels is defined as $L_{E^-} \subseteq L_{E^+}$.

If positive example edges are extracted through provenance extraction algorithms, one could acquire negative example edges by relaxing the set of rules used for inserting the positive edges. For example, if a rule for inserting a long-range influence edge is a maximum difference between node attribute values, a derived negative example sample criteria could use the double of this maximum difference. On the other hand, if positive example edges are provided by human analysis of the graph, additional sample criteria could be, in a similar way, derived from the range of absolute values, differences, average, etc, of attribute's values from nodes connected by positive example edges. It's important to notice that no edge can be in both positive and negative example sets.

GraphSAGE is fed with graphs in the NetworkX² format, JSON files mapping node ids and node classes and a numpy array containing node features (npy file). Node id files map any domain-dependent node id attribute to an integer id value. For example, in provenance graphs, node id is a string. Every provenance node id is then assigned to a unique integer value in the id mapping file. Node class files map the integer id value of a node to a class value. The node class file is used by GraphSAGE for node classification-driven embeddings tasks in a supervised setting. Since our work intends to generate node embeddings in an unsupervised setting, node class file is not relevant to embedding generation. Finally, node features are stored in numpy arrays files. A stored numpy array maps a node id to its respective node features.

Since all the data for these files are stored in the provenance graph data structure, transformation algorithms must be implemented. The provenance graph data is structured as an xml file containing nodes and edges. Every node contains several attributes and its values. Edges contain its connecting nodes ids and pairs of attributes and its values as well. A tree-traversing based algorithm could be used to visit provenance nodes and store id, attributes and class values to its respective files and visit provenance edges and build the graph structure into an instance of a NetworkX graph. Also, GraphSAGE requires that nodes are divided into training, test and validations sets for the supervised setting, as well as edges for unsupervised setting.

²https://networkx.github.io/

```
Algorithm 2: Provenance to GraphSAGE files conversion algorithm
   input : Graph G(V, E), function defineSet : \rightarrow \{"train", "test", "val"\}
   output: NetworkX Graph nx; Id Map id map; Class Map c map; features
            array {\bf f}
1 nx = \{\};
2 id map = \{\};
s c map = \{\};
4 f = [[]];
5 for v \in V do
      original id = GetID(v);
6
      id = length(id map);
7
      id map[id] = original id;
8
      nx.add node(id);
9
      \mathbf{nx}[id]["set"] = defineSet();
10
      for every element el \in v and el is not the ID element do
11
          f[id].append(el.value);
12
          if el is the class element then
13
             c_map[id] = el.value;
\mathbf{14}
15 for e \in E do
      original source, original target = getNodesOriginalID(e);
16
      source = getMapKey(id map, original source);
\mathbf{17}
      target = getMapKey(id map, original target);
18
      nx.addEdge(source, target);
19
```

The pseudocode described in Algorithm 2 covers most requirements to generate Graph-SAGE required files from a provenance graph. Lines 5-14 traverses node elements. Lines 6-8 maps original node ids (regardless of their type) to a unique integer id. Then, the node is added to the NetworkX graph instance nx. In line 10, the current node receives an attribute "set" defined by a function *defineSet*. Lines 11-14 iterate over elements of the current node and append their values to the features array. Since the algorithm iterates over an xml structure, the order in which features are appended to the features array is always the same and must be ordered according to labels of previously defined type t'. The last line of this block assigns an element value for the class mapping function. The class map is not used in the unsupervised setting, therefore, the definition of the class

 $\mathbf{nx}[source, target]["set"] = defineSet();$

 $\mathbf{20}$

element might be ignored. Lines 15-20 traverses edge elements. Connecting node ids are retrieved on line 16 and mapped back from id_map in lines 17 and 18. After the edge is added to nx, it also receives a "set" attribute which value is defined by function *defineSet*. An example of a *defineSet* function could randomly divide elements into train, test and validation using a proportion ratio such as 70%/20%/10%.

In our illustrative scenario, \mathfrak{G} represents all game Y's provenance graphs available. Let the set of node attribute $\mathfrak{t}_{\mathfrak{v}} = (provenance_type, provenance_label, speed, acceleration, direction_x, direction_y, direction_z, angularspeed_x, angularspeed_y, angularspeed_z, timestamp) and the set of node provenance labels types <math>\mathfrak{L}_{\mathfrak{v}} = \{driving_straight, turn-ing_left, turning_right, breaking, crashing, changing_gear, drifting\}$. Bob may select a subset of graphs $\mathfrak{G}' \subset \mathfrak{G}$. Bob annotates influence edges for graphs $G \in \mathfrak{G}'$. These annotated influence edges compose the positive edge example set E^+ , to which belong edges with labels $L_{E^+} = \{driving_straight \to drifting, turning_right \to drifting, turning_left \to drifting, changing_gear \to drifting, drifting \to drifting\}$. In Figure 4.2, sequential edges (black) are automatically extracted. Green dashed edges represent influence edges annotated by Bob. Green edges compose the positive edge example set and are not existing edges in the graph.

Bob implements a method in his favorite programming language that, given all labels L_{E^+} , pairs of nodes with the same labels are found. Bob annotates edges $e'_i \in E$ connecting these pair of nodes if $e'_i \notin E^+$. These second set of annotated edges compose the negative edge examples set $E^- \subset E$. In Figure 4.2, red dashed edges represent negative examples sampled by Bob's method. Notice that the red dashed edge connects two nodes whose labels are similar to a positive example (green dashed edge).

Concerned about the need of employing a neighborhood aggregation approach, Bob compares some edges from positive and negative edge examples set and finds edges $e_k^+ =$ $(v_{k_1}, v_{k_2}) \in E^+$ and $e_m^- = (v_{m_1}, v_{m_2}) \in E^-$ with very similar values for all attributes $\mathfrak{t}_{\mathfrak{v}}$ in the pair of source nodes (v_{k_1}, v_{m_1}) and of target nodes (v_{k_2}, v_{m_2}) , thus rendering it difficult to set a trivial rule for differing positive from negative examples and supporting his decision for a neighborhood aggregation approach. This could be the case of both example edges of type turning_right \rightarrow drifting in Figure 4.2. Finally, Bob writes an algorithm that transforms provenance graphs into files required for GraphSAGE.



Figure 4.2: Example of a game Y's graph during the preprocessing step.

4.4 Embedding generation

As aforementioned, embedding generation is realized using the GraphSAGE framework, which is based on neighborhood aggregation techniques and includes several aggregation methods based on functions and neural network architectures. An exhausting list of aggregation methods implemented within GraphSAGE is:

- 1. Mean-based aggregator: this architecture generates embeddings by taking the elementwise mean of feature vectors.
- 2. LSTM-based aggregator: this architecture generates embeddings by feeding a random permutation of node's neighborhood to an LSTM network.
- 3. Max-pooling aggregator: in this architecture, each neighbor's feature vector is independently fed through a fully-connected neural network. Then, an element-wise max-pooling operation is applied to aggregate information.
- Mean-pooling aggregator: similar to the previous one, but using an element-wise mean-pooling operation.
- 5. GCN-based aggregator: this architecture generates embeddings by feeding node and its neighbors feature vector into an inductive GCN (Graph Convolutional Network).

Using the unsupervised setting of GraphSAGE with any of the provided methods above to aggregate the representation of the nodes, we finally have the embeddings that are going to pose as features to a machine learning classifier. In addition, these embeddings can be used for other downstream machine learning tasks such as clustering and prediction, for example.

In our scenario, Bob should generate node embeddings for all nodes $v \in \mathfrak{G}'$ using a GraphSAGE aggregation technique of his choice. He chooses to generate embeddings using the LSTM aggregator and set the number of output dimension to 128. Now, for each node $v \in \mathfrak{G}'$ exists a node embedding $h_v \in \mathbb{R}^{128}$. Bob could also generate node embeddings using all the aggregation methods and apply some validation technique to compare the performance of each aggregation method. Yet, for conciseness sake, we continue our scenario considering only the LSTM aggregator.

4.5 Classifier training

Classification tasks can be realized using several predictive models, such as decision trees or functions, which in turn can be induced based on optimization algorithms such as stochastic gradient descent. Nowadays, there is a number of different libraries and packages available that implement machine learning algorithms, such as Apache Spark's MLLib[50], SciKit-Learn[16] and WEKA[31]. Any of these classification tools require input examples in the form of features vectors and their respective labels, which are, in our binary case, 0 or 1.

Input feature vectors are composed of example edges encoded through their connecting node's embeddings. In this way, each feature is one of the dimensions of the input vector, represented in a latent vectorial space. Since generated node embeddings are vectors all with the same dimension, it is possible to generate edge embeddings using a simple function $embed_edge(v_1, v_2) = f(x_{v_1}, x_{v_2})$ where v_1 and v_2 are the connecting nodes of edge e and f is an operation such as concatenation, element-wise sum or cross product. Positive examples, representing long-range influence, receive label 1, while negative examples, which are long-range edges that do not represent an influence relation between their connecting nodes, receive label 0.

Finally, the training of the classifier is achieved by feeding the input feature vectors and their classes. The resulting model can then be used to detect the targeted edges in provenance graphs.

In our scenario, Bob chooses SciKit-Learn for the classifier training step. He must then encode input feature vectors from E^+ and E^- by feeding the connecting nodes' embeddings $h_v \in \mathbb{R}^{128}$ for each edge $e_i = (v_k, v_j)$ into a *embed_edge* function of his choice. He chooses to use a concatenation function, which leads to encoded edges $h_e \in \mathbb{R}^{256}$. After that, Bob labels each encoded edges as 1, if a positive example, or 0, if a negative example. Finally, Bob feeds a machine learning algorithm (such as MLP) with input feature vectors and their labels in order to obtain a model M trained to detect the desired type of edge.

4.6 Inference Application

The resulting model M is trained to detect encoded candidate edges. Given a new graph G_n that should be enhanced using model M, all its nodes must be embedded using the same architecture employed in the Embedding generation step. From node embeddings, candidate edges must be sampled. Then, candidate edges must be encoded using the same encoding function employed in the Classifier training step. Encoded edges are fed to the trained model in order to be classified as a target edge or not. If positively classified, candidate edges are added to the graph. Still, strategies for sampling candidate edges from this new graph are also needed, otherwise, one should simply test every possible edge, which is a computationally expensive task.

We list below some advice and strategies about sampling candidate edges for graph enhancement:

- Trivially, it is not necessary to encode candidate edges using nodes already connected in graph G_n , because doing so, one would try to detect edges that already exist in G_n .
- Candidate edges could be encoded by combining nodes whose distance is smaller to *n* neighbors, similar to a BFS-search window approach, excluding the ones that are already connected.
- If a timestamp attribute is available for all nodes, a time window approach could also be applied. In this approach, candidate edges could be encoded by combining nodes whose timestamp difference is smaller than a threshold value *ts*.
- Another approach would be to set node's label constraints such as those defined in the pre-processing step, i.e., candidate edges must connect nodes with label sequences contained in positive example set's node label sequences.
- Both search BFS-window, time and label constraint approaches could be applied simultaneously.

It is worth mentioning that by not using the label constraints approach, the trained model might classify positively candidate edges with node labels differently from the ones defined in positive examples. That means, it is possible that the model M infers an influence between pairs of nodes with label sequence unobserved in the model training. We name this phenomenon *influence discovery*. The aggregation approach and deep learning architectures used in Embedding generation step are believed to be the reason for this phenomenon.

The aggregation methods aggregate information from sampled neighbors. In other words, in these methods, a node's neighborhood affects its resulting embedding. Also, it is well known that deep learning techniques are capable of learning relevant unforeseen features and structures from its inputs. Therefore, candidate edges with unobserved training labels, classified positively by the resulting model M, could also represent existing influences between nodes, similar to the ones used in the training step. Evaluation of this kind of candidate edges could be performed by game analytics and game design experts. Also, we intend to investigate the use of this framework's resulting models on unforeseen influence discovery.



Figure 4.3: Example of a game Y's graph during inference application step.

In our final scenario example step, Bob generates node embeddings for previous game

Y's graphs $\mathfrak{G}'' = \mathfrak{G} - \mathfrak{G}'$. Bob uses both time search window, with ts=2 seconds, and label constraint approaches, that is, candidate edges must be of types $L_{E^+} = \{ driving_straight \}$ \rightarrow drifting, turning_right \rightarrow drifting, turning_left \rightarrow drifting, changing_gear \rightarrow drifting, $drifting \rightarrow drifting$. He feeds encoded candidate edges from all graphs \mathfrak{G}'' to M. In Figure 4.3 we have subgraph from \mathfrak{G}'' containing nodes v_{10} to v_{16} . Sequential edges (colored in black) are edges automatically extracted. Long-range edges are colored in green and red. Candidate edges are (v_{10}, v_{13}) , (v_{11}, v_{13}) , (v_{13}, v_{16}) and (v_{14}, v_{16}) . Edge (v_{11}, v_{14}) is not a possible candidate edge because even though it passes on the time search window, its type turning_right \rightarrow changing_gear $\notin L_{E^+}$. On the other hand, edge (v_{10}, v_{16}) is also not a possible candidate edge because even though its type driving_straight \rightarrow drifting $\in L_{E^+}$, the time difference between these nodes is 2.5s > ts=2. Therefore, model M will only try to infer long-range influence between candidate edges $(v_{10}, v_{13}), (v_{11}, v_{13}), (v_{13}, v_{16})$ and (v_{14}, v_{16}) . Model M receives edges encoded using the pair of nodes of each edge. Green edges represent candidate edges which model M considered as existing influence edges and should, therefore, be added to the final graph. Red edges represent candidate edges rejected by M as existing influences. Therefore, the final version of the subgraph in Figure 4.3 would contain black and green edges.

At the end of the process, Bob has a new set of graphs enhanced with new influence edges. Also, forthcoming graphs can also be processed by the influence edge detector, so that Bob can resume his game analytic tasks with more complete working material.

4.7 Final considerations

In this chapter, we presented the PingUMiL, a framework for detecting long-range causeand-effect relationship by leveraging methods of representation learning in game provenance graphs and explained in details all steps and tools employed by the framework. The PingUMiL framework is defined by four steps: graph capture, pre-processing, embedding generation and classifier training. Also, we also explained how a PingUMiL model could be used in a real world scenario. For better understading of each step, a guiding example was used to contextualize the framework.

In the next chapter, we present experimental methodology and results, by presenting a materialization of the PingUMiL framework on two case studies.

Chapter 5

Experimental Results

This chapter presents our experimental methodology and results. Section 5.1 illustrates our proposal applied in two racing game prototypes, detailing every step of the framework. After that, the results from experiments realized in order to evaluate research objectives defined in the Introduction chapter are presented in section 5.2.

5.1 Case Study

Both game prototypes are example stages of racing games in which a single player drives along a single track. The first game is Car Tutorial Unity $(CT)^1$, presented in Figure 5.1(a), and the second one is Arcade Car Physics $(AC)^2$, presented in Figure 5.1(b). Car Tutorial Unity (CT) is a free prototype asset for racing games, designed for Unity 3.x, i.e., an older version of the game engine. The game is single player, contains only a single track and uses Unity's native car physics. Arcade Car Physics (AC) is an open source prototype implemented in Unity 2018. Like CT, this prototype is single player and has only one track, even though it provides several objects outside the track, like bridges and ramps, for simulating physics. However, this prototype presents a significant difference from CT when it comes to the car game entity's physics. AC implements several algorithms over or instead native engine physics, such as Speed Curve, Ackermann Steering and Stabilizer Bar Forces[49].

Beyond serving as examples for the PinGUMiL framework application, both case studies are used for evaluating the secondary objectives stated in the introduction. That is, all graphs, edge sets and models produced by the steps described in the following

¹https://assetstore.unity.com/packages/templates/tutorials/car-tutorial-unity-3-x-only-10 ²https://github.com/SergeyMakeev/ArcadeCarPhysics





(a) Car Tutorial Unity screenshot. Source: https://answers.unity.com/ questions/582986

(b) Car Arcade Physics screenshot.

Figure 5.1: Racing games screenshots.

sections are used in our Results section (5.2).

5.1.1 Graph capture

The first step is to capture provenance graph from game sessions. The provenance extraction algorithm for CT developed by Kohwalter et al. is used in CT[40] and adapted with minor changes for AC, mostly due to differences in physics implementation between both games.

Both games were made available for 4 playtesters, all male, age between 20 and 30 years old, experienced with racing games and considered themselves hardcore gamers. All instructions given to playtesters were:

- Play the game as much as they wanted.
- Play the game as a racing game, completing at least one lap.
- At the end of the game session, send the output xml file.

For CT, 10 game session graphs were extracted, which in total contain 9194 nodes and 47497 edges. For AC, 3 game session graphs were extracted, which in total contain 4146 nodes and 21397 edges. The number of game sessions per game was influenced solely by playtesters availability. Notice that 3 AC graphs have almost half the number of nodes and edges of 10 CT graphs, since the amount of recorded data is proportional to the game session duration.

5.1.2 Preprocessing

Provenance extraction algorithm for CT[40] already implemented long-range influence edges capture for the following type of edges: Crash \rightarrow Crash, Crash \rightarrow LostControl, Crash \rightarrow Scraped, Flying \rightarrow Crash, Flying \rightarrow Landing, Flying \rightarrow LostControl, Flying \rightarrow Scraped, HandBrake \rightarrow LostControl, Scraped \rightarrow Crash, Scraped \rightarrow Scraped. Therefore, the positive examples were automatically extracted and removed from the graph in this step for both CT and AC.



Figure 5.2: Flying \rightarrow Crash edges examples.

Once the positive example set is defined, it is possible to define the sample criteria for negative edge examples generation. The first sample criteria is the set of types of positive examples. In this case, negative examples would necessarily connect two nodes with labels represented in the positive example set, respecting the node label's order. For better understanding, a positive and negative example of Flying \rightarrow Crash edges are shown in Figure 5.2. For both examples, the nodes are colored according to the player's speed value, i.e. the higher the speed the higher the color saturation. The positive example shown in Figure 5.2(a) presents an influence edge between nodes 74 (Flying) and 81 (Crash). In this example, the player flies, lands, and tries to prevent a crash by not accelerating, which results in a sequence of ChangedGear nodes. Still, he is not able to prevent the crash. A case of a candidate negative example is shown in Figure 5.2(b). Notice that the edge between node 115 (Flying) and node 118 (Crash) is a positive example. Node 118 (Crash) leads to another Crash node, i.e. node 119. An edge between nodes 115 (Flying) and 119 (Crash) is a negative example, since Crash in node 119 is a direct consequence from Crash node 118.

The second criteria is derived from the provenance extraction algorithm. Kohwalter et al. define a time constraint [40] of two seconds for non-consecutive sequential nodes with most label sequences defined in the first criteria. We define a relaxed constraint based on node path distances by using a search window of 10 nodes. Statistics regarding all positive and negative examples for CT an AC are shown in Tables 5.1 and 5.2 respectively.

Edge Type	Positive Examples	Negative Examples
$\operatorname{Crash} \to \operatorname{Crash}$	98	138
$\mathrm{Crash} \to \mathrm{LostControl}$	16	230
$\operatorname{Crash} \to \operatorname{Scraped}$	3	150
$Flying \to Crash$	199	266
$Flying \rightarrow Landing$	42	507
$Flying \rightarrow LostControl$	90	343
$Flying \to Scraped$	1	265
${\rm HandBrake} \rightarrow {\rm LostControl}$	62	68
Scraped \rightarrow Crash	81	78
Scraped \rightarrow Scraped	2	99
Total	594	2144

Table 5.1: Total number of CT's positive and negative examples.

Table 5.2: Total number of AC's positive and negative examples.

Edge Type	Positive Examples	Negative Examples
$\operatorname{Crash} \to \operatorname{Crash}$	38	31
$\operatorname{Crash} \to \operatorname{LostControl}$	11	85
$\operatorname{Crash} \to \operatorname{Scraped}$	6	127
$Flying \rightarrow Crash$	51	125
$Flying \to Landing$	136	278
$Flying \rightarrow LostControl$	29	165
$Flying \rightarrow Scraped$	8	217
$\mathrm{HandBrake} \rightarrow \mathrm{LostControl}$	4	15
Scraped \rightarrow Crash	56	38
Scraped \rightarrow Scraped	11	153
Total	350	1234

Since edge types distribution is unbalanced between negative and positive examples, both sets are reduced so that every edge type has the same amount of examples in both positive and negative sets. For this balancing, edges are chosen randomly. Hence, positive and negative sets will also have the same number of examples. The resulting datasets for both games are shown in Table 5.3.

Edge Type	CT Examples	AC Examples
$\operatorname{Crash} \to \operatorname{Crash}$	196	62
$\operatorname{Crash} \to \operatorname{LostControl}$	32	22
$\operatorname{Crash} \to \operatorname{Scraped}$	6	12
$\mathrm{Flying} \to \mathrm{Crash}$	398	102
$Flying \rightarrow Landing$	84	272
$Flying \rightarrow LostControl$	180	58
$Flying \rightarrow Scraped$	2	16
$\mathrm{HandBrake} \rightarrow \mathrm{LostControl}$	124	8
Scraped \rightarrow Crash	156	76
Scraped \rightarrow Scraped	4	22
Total	1182	650

Table 5.3: Total number of edge examples for both CT and AC.

5.1.3 Embedding generation

Embedding generation is realized using GraphSAGE on its unsupervised setting. The default settings were applied for most architectures, except for using 10 epochs, number of samples in layers 1 and 2 sample_1 = sample_2 = 2 and the number of output dimensions in the first layer dim_1=256. These settings were observed to reach minimal error during embedding generation. We generated the embeddings with all available architectures.

Each architecture performance and their resulting embeddings are evaluated and discussed in section 5.2 through several experiments and simulations.

5.1.4 Classifier training

Classifier training is realized using positive and negative examples sets defined in the preprocessing step and node embeddings defined on the previous step. We adapted the node classification evaluation algorithms provided by GraphSAGE so that it takes encoded edges as inputs and their labels as output. As aforementioned, edges are encoded using connected node's embeddings. We used concatenation to encode node embeddings into edges. After preparing the data for the classifier training, we opted for sci-kit learn [16], a well known Machine Learning library in python which provides several supervised learning models based on many well-known solutions (linear models, SVM, Decision Trees and Neural Networks).

Also, we discuss the performance of classifier models when detecting edges using embedded data for both AC and CT graphs through several experiments and simulations in the following section.

5.2 Results

In this section, we describe how PingUMiL corroborates the proposed research objectives through the experiments performed on case studies described in section 5.1. As a matter of convenience, the aforementioned research objectives are:

- evaluate if graph representation learning improves the detection of influence edges compared to classical machine learning techniques with raw features;
- evaluate if the framework achieves generalization capability that allows inferring indirect influence edges in unobserved but similar provenance graphs.

We test the performance of PingUMiL generated classifier models for AC and CT. There are three main settings which describe a model: aggregation architecture (used in embedding generation step), edge encoding function (defined in classifier training step) and classifier method (defined in inference application step). In our experiments, every model uses a combination of the following options:

- Aggregation architecture: LSTM, MaxPool, MeanPool, Mean, GCN
- Edge encoding functions: Mult(Elementwise Multiplication) and Cat (Concatenation)
- Classifier method: MLP (multilayer perceptron neural network classifier), SGD (Stochastic Gradient Descent based classifier), SVM (Support Vector Machine based classifier).

For example, a model PingUMiL.LSTM + Concat + MLP(100,1) is a model which relies on the LSTM aggregator for embedding generation, concatenation function for edge encoding and an MLP neural network whose architecture is described by a tuple (k,n)where the k represents the number of hidden units per layer and n represents the number of hidden layers, i.e., MLP(100,1) is an MLP with a single hidden layer with 100 units.

In our experiments, we measure precision, recall and f1-score [54]. Precision represents how accurate a classifier is regarding their positive predictions and is defined as

$$Precision = \frac{tp}{tp + fp}$$

where tp represents the number of true positives, i.e. positives edges that were classified correctly and fp represents the false positives, i.e. negative edges that were classified as positive edges. In the provenance graph context, precision metrics give an insight on the quality of PingUMiL model's positively classified example edges.

Recall, in its turn, represents how accurately a classifier reacts to a positive example and is defined as

$$Recall = \frac{tp}{tp + fn}$$

where fn represents the number of false negatives, i.e. positive edges that were classified as negative edges. In the provenance graph context, recall metrics determine how often a PingUMiL model correctly classifies a positive example edge.

Precision, recall and f1-score are measured for both study cases encoded edges in a stratified k-fold cross-validation setting [34]. That means, encoded edges are split into k folds. One fold is taken as the test data set while the remaining ones are taken as the training data set. Then, a model is fit on the training set and evaluated on the test set. This procedure is realized k times so that every fold is used as test set once. In our results, we show the average mean of all folds measured metrics. Regarding k-fold cross-validation of our study cases, CT is divided into 7 folds while AC is divided into 4 folds.

We compare the generated classifiers from PingUMiL against traditional classification methods using the raw features and the same classifiers as before (SVM, MLP, and SGD). We call raw features the attribute values attached to each node in the original provenance graph, such as speed, acceleration, position, etc. The raw node features were fed to these methods in order to provide a baseline. The main motivation of this baseline is to investigate whether the use of embeddings enhance edge detection over raw features and evaluate the secondary objective 2. We also use these experiments to evaluate secondary objective 1 since both classical and graph representation approaches are applied for influence detection.

Finally, we investigate the generalization among case studies by observing the performance of a model trained with CT encoded edges and tested with AC encoded edges. We intend to check whether it is possible to enhance a graph from a game using a previously trained model from a similar game. This experiment evaluates secondary objective 3. We also provide a baseline for this experiment using raw node features.

5.2.1 Car Tutorial

As aforementioned, Car Tutorial example edges were split into 7 folds and fed into a number of different learning settings, leading to more than **60** models. Tables 5.4 and 5.5 presents averaged time duration of embedding generation (for all CT nodes) and some classifiers' training step (per fold), respectively. Each combination of fold and model is trained 50 times and their performance is measured and averaged for each fold and for each model. Table 5.6 presents some results for generated models' analysis. All averaged metrics presented in Table 5.6 has variance < 0.002. The best average performance was achieved by LSTM-based aggregation method, concatenation as edge encoder and an MLP neural network with approximately 67% on all metrics, which implies in a 13% gain over the best baseline.

Table 5.4: Averaged time (seconds) duration of CT's embedding generation step.LSTM MaxPool MeanPool Mean GCN61.23625.82324.87622.36720.175

Tuble 0.0, Theraged unit (becomdb) duration of CT b clubbing for the	Table 5.5 :	Averaged	time	(seconds)	duration	of	CT's	classifier	training	per	fol	d.
--	---------------	----------	------	-----------	----------	----	------	------------	----------	-----	-----	----

Classifier	LSTM	MaxPool	MeanPool	Mean	GCN
MLP(100,1)	3.811	4.912	3.104	2.577	3.033
MLP(100,2)	2.504	3.109	3.021	1.9	2.327
MLP(256,1)	6.802	8.3	6.903	4.922	5.062
SGD	0.128	0.122	0.122	0.125	0.12

Table 5.6: Averaged results of CT classifiers in a 7-fold cross-validation setting.

Approach	Precision	Recall	F1
Raw features + SVM	0.249	0.293	0.261
Raw features $+$ MLP	0.381	0.382	0.381
Raw features $+$ SGD	0.545	0.542	0.534
$\operatorname{PingUMiL.LSTM+Mult+MLP(100,1)}$	0.673	0.663	0.664
$\operatorname{PingUMiL.LSTM+Mult+MLP}(100,2)$	0.641	0.654	0.638
$\operatorname{PingUMiL.LSTM+Cat+MLP(100,1)}$	0.664	0.662	0.661
$\operatorname{PingUMiL.LSTM+Cat+MLP(512,1)}$	0.672	0.656	0.662
$\operatorname{PingUMiL.LSTM+Cat+MLP(100,2)}$	0.674	0.668	0.669
PingUMiL.MeanPool+Cat+SGD	0.605	0.598	0.575
PingUMiL.Mean+Cat+MLP(100,1)	0.613	0.618	0.614
${\rm PingUMiL.GCN}{+}{\rm Mult}{+}{\rm MLP}(100,\!2)$	0.549	0.567	0.538

We observed that the aggregator architectures Mean and GCN tend to score less than the other architectures on several experimental settings for CT. Also, in some settings, models composed of these architectures presented metrics lower than baseline. Analogous trend holds for the classifier methods SVM and SGD. Therefore, it is possible that these architectures and methods are not suitable for the intended task with CT graphs.

LSTM, MaxPool and MeanPool aggregation architectures, on the other hand, have shown more than 10% gain over baseline with MLP based classifiers. Results show little variation between edge encoding functions and MLP architectures with these aggregation architectures. Still, the most relevant results were achieved with the number of hidden layers n between 1 and 10 and hidden units per layer k between 100 and 512. The best average precision inside a fold was achieved by PingUMiL.LSTM + Mult + MLP(100,1) with 72,3%, while the best average recall and F1 was achieved by PingUMiL.LSTM + Cat + MLP(100,1) with 72,2% and 71,9% respectively. These top performances suggest that PingUMiL generated models capable of detecting target edges. We expect similar results for AC experiments.

Another relevant observation about the aforementioned top performances is that they were achieved on the same fold (fold 1). Analogously, several models performed better on fold 1 than on the other folds. It is possible that fold 1's edge sets have some characteristic or pattern which improves classifier training.

After evaluating the overall performance of the generated models, an in-depth investigation regarding types of nodes was realized. For this investigation, the output of a generated PingUMiL.LSTM + Cat + MLP(100,1) classifier for each test fold is taken randomly. After that, precision and recall metrics are calculated per edge type per fold. Table 5.7 presents the mean average and variance of these metrics.

Tuble 5.1. Theraged performance results per suge for C1.							
Edge Type	Avg. Precision (Var)	Avg. Recall (Var)					
$\operatorname{Crash} \to \operatorname{Crash}$	$0.686\ (0.013)$	0.712(0.021)					
$\operatorname{Crash} \to \operatorname{LostControl}$	1(0)	$0.881 \ (0.043)$					
$Flying \rightarrow Crash$	0.72(0.006)	$0.796 \ (0.006)$					
$Flying \rightarrow Landing$	0.853(0.041)	$0.81 \ (0.022)$					
$Flying \rightarrow LostControl$	0.643(0.003)	0.724(0.015)					
$\mathrm{HandBrake} \rightarrow \mathrm{LostControl}$	$0.647 \ (0.012)$	0.727 (0.028)					
Scraped \rightarrow Crash	$0.714\ (0.029)$	$0.681 \ (0.026)$					

Table 5.7: Averaged performance results per edge for CT

Edges of type Crash \rightarrow Scraped, Flying \rightarrow Scraped and Scraped \rightarrow Scraped are not present in Table 5.7 due to their low occurence in CT graphs. As shown in Table 5.3, the number of examples (positive and negative altogether) for these 3 types of edges are 6, 2 and 4 respectively. For this reason, some folds did not contain examples of these edges. Results in Table 5.7 shows that the best average precision and recall was achieved in edges of type Crash \rightarrow LostControl, Flying \rightarrow Landing and Flying \rightarrow Crash. It's important to take into account the wide difference between the number of examples per type of edge when analyzing the Table 5.7. Edge type with the lowest number of examples in this experiment is Crash \rightarrow LostControl with 32 edges, while the highest is Flying \rightarrow Crash with 398 edges. Notably, the model achieved the highest performance metric for these edge types. Regarding results with Crash \rightarrow LostControl, it is not safe to assume that this performance would hold in a real-world scenario due to the low number of example edges. Beyond that, a similar result in AC could support Crash \rightarrow LostControl high performance. Flying \rightarrow Lading and Flying \rightarrow Crash results, on their turn, show that the model could differ true and false indirect edges with high performance.

It's possible to observe in the Figure 5.3, in which the horizontal axis lists edge types and the vertical edges represents the number of edges predicted as FN (false negatives, true indirect edges classified as false indirect edges), FP (false positives, i.e. false indirect edges classified as true indirect edges), TN (true negatives, false indirect edges classified as false indirect edges) and TP (true positives, true indirect edges classified as true indirect edges), that a trend holds for most edge types: most edges are correctly classified (TP and TN outnumbers FP and FP) and the number of false positives is higher than the number of false negatives.

In our understanding, false negatives are more dangerous to game analytics tasks than false positives. In a real-world scenario, a false negative would be a true influence edge incorrectly classified and, therefore, not inserted in the graph enhanced by PingUMiL. On the other hand, a false positive would insert an influence edge that (1) makes no sense in its neighborhood context and could be ignored by the game analyst or (2) represents an unforeseen influence with regard to edge examples fed to PingUMiL model. That last possibility would need to be validated by the game analyst and could lead to refinement of the model and discovery of new influences edge types.

5.2.2 Arcade Car

The Arcade Car example edges were split into 4 folds and fed into the same learning settings used in CT experiments. Tables 5.8 and 5.9 presents averaged time duration of embedding generation (for all AC nodes) and some classifiers' training step (per fold), respectively. Similarly, each combination of fold and model is trained 50 times and their performance is measured and averaged for each fold and for each model. Table 5.10



CT Overall classifier results per edge type

Figure 5.3: Overall classification results per edge type in CT graphs.

presents some results for generated models' analysis. All averaged metrics presented in Table 5.6 had variance < 0.002. The best average performance was obtained by LSTM-based aggregation method, concatenation as edge encoder and an MLP neural network with approximately 70% on all metrics, which implies in a 10% gain over the best baseline.

Table 5.8 :	Averaged tin	me (seconds)	duration of A	AC's embe	edding g	eneration	step
	LSTM	MaxPool	MeanPool	Mean	GCN	_	

20.936

58.839

24.061

18.328

17.091

Table 5.9: Averaged time	(seconds)	duration	of AC's	classifier	training	per fold.

Classifier	LSTM	MaxPool	MeanPool	Mean	GCN
MLP(100,1)	4.719	4.126	4.501	4.467	4.831
MLP(100,2)	2.117	1.872	1.965	1.839	2.517
MLP(256,1)	7.726	6.445	6.656	6.08	8.176
SGD	0.131	0.148	0.14	0.138	0.144

Regarding the aggregation architecture, the highest performances were again achieved using LSTM. However, different from CT experimental results, the Mean and GCN aggregators achieved a better performance than the baseline classifiers in several experiments, while MaxPool aggregator presented lower performance compared to the baseline methods.

Approach	Precision	Recall	$\mathbf{F1}$
Raw features $+$ SVM	0.249	0.334	0.273
Raw features $+$ MLP	0.394	0.394	0.394
${\rm Raw\ features\ +\ SGD}$	0.596	0.578	0.575
$\operatorname{PingUMiL.LSTM+Cat+MLP(100,1)}$	0.696	0.7	0.697
$\operatorname{PingUMiL.LSTM+Cat+MLP(100,2)}$	0.695	0.69	0.692
$\operatorname{PingUMiL.LSTM+Cat+MLP}(256,1)$	0.696	0.702	0.698
$\operatorname{PingUMiL.LSTM+Cat+SGD}$	0.662	0.613	0.608
PingUMiL.MeanPool+Mult+MLP(100,1)	0.646	0.638	0.641
$\operatorname{PingUMiL.Mean+Cat+MLP(100,1)}$	0.665	0.628	0.646
$\operatorname{PingUMiL.GCN+Cat+MLP(256,1)}$	0.695	0.663	0.678

Table 5.10: Averaged results of AC classifiers in a 4-fold cross-validation setting.

Notice that most of the results in the Table 5.10 use the Cat (Concatenation) edge encoding function. Most models using Mult (elementwise multiplication) encoding function performed worse than its concatenation counterparts. Results for the model PingU-MiL.MeanPool + Mult + MLP(100,1) achieved the highest performance using a model with the Multi edge encoding function.

The performance reached by MLP classifiers were analogous to the previously discussed CT experiments, except for the use of 256 hidden units, which tended to outperform other MLP configurations. The best average precision and F1 inside a fold were achieved by PingUMiL.GCN + Cat + MLP(256,1) with 77,1% and 74,3%, respectively, on fold 4, while the best average recall was achieved by PingUMiL.MeanPool + Cat + MLP(256,1) with 74% on fold 4. These measures corroborate the edge detection capability of PingUMiL models. Similar to fold 1 in CT experiments, folds 1 and 4 concentrated best performance metrics in their experiments.

Observing the results presented in both Tables 5.6 and 5.10, it is possible to conclude about secondary objectives 1 and 2. For secondary objective 1, machine learning techniques are able to detect influence between game components represented as edges in a game provenance graph, given that the models achieved above 77% averaged precision. For secondary objective 2, PingUMiL best performance presents a gain of at least 10% over classical machine learning approaches.

Similar to the previous section, we investigate model's performance regarding type of nodes using a generated PingUMiL.GCN + Cat + MLP(256,1) classifier for each test fold. Precision and recall metrics are calculated per edge type per fold. Table 5.7 presents the mean average and variance of these metrics.

Table 9.11. Avera	Table 5.11. Averaged performance results per edge for AC.						
Edge Type	Avg. Precision (Var)	Avg. Recall (Var)					
$\operatorname{Crash} \to \operatorname{Crash}$	0.59(0.045)	$0.645\ (0.047)$					
$\operatorname{Crash} \to \operatorname{LostControl}$	$0.855\ (0.029)$	$0.9175\ (0.027)$					
$Flying \rightarrow Crash$	$0.635\ (0.004)$	$0.645\ (0.003)$					
$Flying \rightarrow Landing$	$0.7325 \ (0.007)$	$0.575\ (0.001)$					
$\mathrm{Flying} \to \mathrm{LostControl}$	0.915(0.011)	$0.87 \ (0.032)$					
Scraped \rightarrow Crash	0.58(0.015)	$0.683 \ (0.023)$					
Scraped \rightarrow Scraped	0.937 (0.015)	0.835(0.036)					

Table 5.11: Averaged performance results per edge for AC.

Edges of type Crash \rightarrow Scraped, Flying \rightarrow Scraped and HandBrake \rightarrow LostControl are not present in Table 5.11 due to their low occurrence in CT graphs (< 20). As shown in Table 5.3, the number of examples (positive and negative altogether) for these 3 types of edges are 12, 16 and 8 respectively.

The results in the Table 5.11 shows that the best average precision and recall was achieved in edges of type Crash \rightarrow LostControl, Flying \rightarrow LostControl and Scraped \rightarrow Scraped, with number of examples equals to 22, 58 and 22 respectively. In other words, Crash \rightarrow LostControl, Flying \rightarrow LostControl and Scraped \rightarrow Scraped are the edge types with the lowest number of examples in this experiment. Similarly to results shown for CT in Table 5.7, edge type Crash \rightarrow LostControl achieve precision and recall above 80%. We believe that models from both games learned to classify correctly this type of edge due to the low difference among true influence edges which, in its turn, derives from the low occurrence of this type of edge. The same intuition holds for Flying \rightarrow LostControl and Scraped \rightarrow Scraped in this experiment. Different from CT, the edge type with most examples, Flying \rightarrow Lading, achieved low recall performance (57,5%).

Figure 5.3 shows a bar graph in which the horizontal axis lists edge types and the vertical axis represents the number of edges predicted as FN, FP, TN and TP. Different from results in CT, AC presents the smallest gap between falsely classified examples (FN and FP) and correctly classified examples (TN and TP). In Flying \rightarrow Landing, Flying \rightarrow LostControl and Scraped \rightarrow Scraped edges, the number of TN is higher than TP, i.e. the models managed to correctly classify false examples easier than true ones. Consequently, the number of FN is higher than FP, which is dangerous for the reasons already discussed on the previous subsection: a false negative edge would be a true influence edge incorrectly classified and, therefore, not inserted in the graph enhanced by PingUMiL, which leads to missing information to game analytics tasks.



AC Overall classifier results per edge type

Figure 5.4: Overall classification results per edge type in AC graphs.

5.2.3 Generalization among racing games

In this experiment, we try to evaluate secondary objective 3. We observe the generated models' generalization capacity over different, yet similar, games by comparing the performance of the models trained with the AC game against two other models: (1) one trained with the dataset generated from CT and tested to detect AC edges, identified with a CT prefix; and (2) one trained with both CT and AC, identified with CTAC prefix, and tested to detect AC edges. In this latter case, each AC training fold is concatenated with a CT training fold. In both cases we choose AC to test the model because it has fewer examples than CT. Similar to the previous experiments, AC edges are split into 4 folds, the models are trained 50 times and their performance are measured and averaged for each fold and for each model. Time performance was also measured for some CTAC models. Tables 5.12 and 5.13 presents averaged time duration of embedding generation (for all CTAC nodes) and some classifiers' training step (per fold), respectively.

Table 5.12: Averaged time (seconds) duration of CTAC's embedding generation step.LSTM MaxPool MeanPool Mean GCN64.3230.06526.74123.50227.361

We propose tests with the first model setting in order to measure how a model trained

Classifier	LSTM	MaxPool	MeanPool	Mean	GCN
MLP(100,1)	6.424	6.125	6.569	5.485	5.123
MLP(100,2)	4.097	3.917	3.796	4.031	4.064
MLP(256,1)	11.207	9.2	8.66	8.186	9.841
SGD	0.356	0.319	0.341	0.328	0.335

Table 5.13: Averaged time (seconds) duration of CTAC's classifier training per fold.

only with CT edges responds to AC edges. On the other hand, the intuition behind the test with the second model setting is that a model performance metrics shall increase when it learns from a larger dataset. But, in this case, the increment in the training data derives from a similar yet different game. We also provide a baseline using raw features and traditional classification methods, similarly to the previous experiments. The models with the highest metrics scores for each model setting in this experiment are shown in Table 5.14.

Table 5.14: Mean averaged results of the generalization experiments.

Approach	Precision	Recall	$\mathbf{F1}$
CT-Raw features + MLP	0.46	0.457	0.461
CT-Raw features + SGD	0.554	0.548	0.534
CTAC-Raw features + MLP	0.596	0.588	0.592
CTAC-Raw features + SGD	0.583	0.571	0.577
CT-PingUMiL.LSTM+Cat+MLP(100,1)	0.531	0.408	0.461
CT-PingUMiL.LSTM+Mul+SVM	0.533	0.629	0.573
CTAC-PingUMiL.LSTM+Cat+MLP(256,1)	0.703	0.693	0.697

The results show that CT-PingUMiL.LSTM + Mult + SVM model achieved a slightly better performance on recall metric, while best precision metric was achieved by CT-Raw features + SGD. This experiment indicates that the generalization over different, yet similar, games is not guaranteed, at least when using exactly the model trained with one game to directly test on another game, without any further training. The reason behind this is that AC and CT node embeddings generated through GraphSAGE's mapping of node features into latent representational spaces might learn different parameters for each game. That is, the AC test set seems to be located into a different distribution space than the CT set.

CTAC-PingUMiL.LSTM+Cat+MLP(256,1) achieved the best performance in this experiment: 70,3% precision, 69,3% recall and 69,7% f1-score. The results are very similar to the ones obtained using only AC edges for training and test shown in Table 5.10. The difference between metrics are close to 1%. Similar trend holds for best average performance into a fold: CTAC-PingUMiL.MaxPool+Cat+MLP(256,1) achieved 73,3% precision on fold 3, while CTAC-PingUMiL.LSTM+Cat+MLP(256,1) achieved 74,7% recall and 72,8% F1 on fold 4. We can assume that the models did not benefit from using CT training edges to detect AC edges on overall performance statistics. We proceed to investigate performance for each edge type, similar to previous subsections 5.2.1 and 5.2.2.

We investigate model's performance regarding type of nodes using a generated PingU-MiL.LSTM + Cat + MLP(256,1) classifier for each test fold. Precision and recall metrics are calculated per edge type per fold. Table 5.15 presents the mean average and variance of these metrics.

Table 5.15. Averaged performance results per edge for CTAC.						
Edge Type	Avg. Precision (Var)	Avg. Recall (Var)				
$\operatorname{Crash} \to \operatorname{Crash}$	0.58 (0.017)	0.61 (0.011)				
$\mathrm{Crash} \to \mathrm{LostControl}$	0.793(0.021)	1(0)				
$Flying \rightarrow Crash$	$0.643 \ (0.014)$	$0.73 \ (0.037)$				
$Flying \rightarrow Landing$	0.673(0.014)	$0.655\ (0.012)$				
$\mathrm{Flying} \to \mathrm{LostControl}$	$0.855\ (0.015)$	$0.835\ (0.025)$				
Scraped \rightarrow Crash	0.605(0.006)	$0.723\ (0.075)$				
Scraped \rightarrow Scraped	$0.937 \ (0.016)$	0.833(0.112)				

Table 5.15: Averaged performance results per edge for CTAC.

Performance metrics presented in Table 5.15 were measured over the same edges of Table 5.11. Therefore, it is interesting to compare results in both tables and relate it to number of examples per edge type on the CT examples set.

Edges of type Crash \rightarrow Crash and Scraped \rightarrow Scraped achieved very similar performance. This result is expected for Scraped \rightarrow Scraped edges because CT example set contains only 4 edges of this type. On the other hand, Crash \rightarrow Crash added 196 examples (thrice the number of examples in AC) in the model's training phase and that, as results points, had no effect on precision and recall performances.

Edges of type Crash \rightarrow LostControl and Flying \rightarrow Landing presented gain on recall metrics (7% and 8% respectively) and a decrease in precision metrics (approximately 6% for both). CT training edges add 32 Crash \rightarrow LostControl examples and 84 Flying \rightarrow Landing examples. These results means that edges from CT enhanced the model's comprehension of what defines a negative example at the cost of difficulting comprehension of what defines a positive example. Since we believe a high recall value is more essential for provenance graph enhancement, this result shows that for some type of edges, a model generated using different games might lead to improvements in preferred metrics.

Edges of type Flying \rightarrow Crash and Scraped \rightarrow Crash presented gain on both recall

(1% and 2.5% respectively) and precision (8.5% and 4% respectively) metrics. For both edge types, the number of CT examples is more than twice the number of AC examples. This result corroborates to a positive answer for secondary objective 3, in which the use of another similar game data improves the capacity to infer long-range influences.

Edges of type Flying \rightarrow LostControl presented loss on both precision and recall metrics (6% and 3.5% respectively). Similar to Flying \rightarrow Crash and Scraped \rightarrow Crash, the number of CT examples is more than twice the number of AC examples. The difference lies in the fact that CT models achieved lower performance metrics on Flying \rightarrow LostControl (64.3% and 72.4%) then AC models. The insertion of new CT edges examples prevented the CTAC model to achieve the same capability that AC example models seem to provide. For this type of edges, it would be best to rely only on AC edge examples.

Figure 5.5 shows a bar graph in which the horizontal axis lists edge types and the vertical axis represents the number of edges predicted as FN, FP, TN and TP. Even though overall performance metrics remain the same between AC and CTAC models, it is possible to notice that Figure 5.5 corrects the distribution of FN, FP, TN and TP classified edges presented in Figure 5.4, i.e. the number of false negatives is the lowest in almost all edges (except for Scraped \rightarrow Scraped). On the other hand, the overall number of true negative and true positive examples decreases when compared to AC results, mainly in the edge type with most examples (Flying \rightarrow Landing).

Finally, we conclude from this generalization across games experiment that simply feeding new edges from a similar game doesn't guarantee overall model's quality enhancement for the racing games used in the experiment. However, PingUMiL models could be enhanced by fine-tuning the model with some types of edges, especially edges with fewer examples and/or lower accuracy performance. Further investigations about quality enhancement through generalization using other types of games is suggested as future work.

5.3 Threats to Validity

The previously described experiments and their respective results present three main validity threats. The first threat lies in the extraction process of provenance graphs, when several gameplay sessions were recorded. It's possible that players behavior during gameplay session could differ from their normal behavior when playing racing games, since both games are prototypes and the sole purpose of the gameplay sessions was to support



CTAC Overall classifier results per edge type

Figure 5.5: Overall classification results per edge type in CTAC experiment setting.

this research.

Another threat lies on baselines derived from classical machine learning. All experiments with these techniques were realized with default settings. Therefore, it's possible that a fine-tuned classical machine learning technique provides better results than the baselines presented in this work.

Also, number of folds for both CT and AC were defined so that each fold from both games had approximately the same number of examples. Therefore, a different experimental setup, where the number of folds and the amount of examples per fold were not related between games, could lead to different insights and conclusions.

Finally, precision and recall analysis per edge type performed on subsections 5.2.1, 5.2.2 and 5.2.3 were realized using generated PingUMiL models chosen at random. Therefore, it's possible that presented results belong to an outlier model with above or below average accuracy performance.

Chapter 6

Conclusions

This work, to the best of our knowledge, is the first attempt in the literature to combine a representation learning and game provenance. We introduced PingUMiL, a framework for enhancing game provenance based on graph-based representation learning, motivated by a long-range indirect edge detection task. PingUMiL includes four steps for running edge detection tasks on game provenance graphs: graph capture, pre-processing, embedding generation, and classifier training. These four steps generate a model which can then be applied to detect missing edges and graph enhancement. We found that edge detection, especially long-range influence edges, is possible using both classic and graph representation learning based machine learning approaches. Experiments were conducted on two racing games and, for both approaches, precision and recall metrics in several experimental settings scored above 50%. Still, models generated by PingUMiL have achieved better performance than classical machine learning techniques with raw features with a gain of at least 10% on precision, recall, and F1 scores. Finally, by using a CT and AC-trained model for classifying AC edges we could verify that even though the insertion of new edges from CT did not enhance model's overall performance, it did enhanced precision and recall metrics for some edge types. This conclusion suggests further investigation towards fine-tuning of PingUMiL models. Also, experiments using other types of games are also suggested in order to corroborate our conclusions about PingUMiL models.

During the experiments, some combinations of GraphSAGE's aggregator architectures and classifier approaches have proven to be unsuitable for the task at hand due to their poor performance. Since PingUMiL is a general framework and defines a set of steps for edge detection tasks, any tool used in our experiments can be substituted in the future. For example, we intend to investigate the use of other embedding generation techniques in near future. Even though GraphSAGE is a powerful, usable and expressive embedding generation tool, it still lacks heterogenous nodes support. It is important to mention that the PingUMiL framework should benefit from upcoming improvements and advances in graph representation learning techniques.

Other possible future works are related to prediction tasks and reinforcement learning. For a prediction task, PingUMiL would be used in a scenario where the complete graph is not available. In this scenario, PingUMiL should be capable of predicting future nodes and the edges that connect predicted nodes to existing ones. Regarding reinforcement learning, we suggest that a learned representation for game provenance graphs, subgraphs and nodes could be used for game enhancement and agent training.

Also, PingUMiL could also be easily adapted for provenance graphs from other domains such as scientific workflow and software engineering.

In conclusion, our results suggest that PingUMiL can be a useful tool for game analytics tasks envolving game provenance graphs such as long-range influence edge detection. We believe that other several game analytics tasks can also be attacked by PingUMiL generated models by performing minor adaptations on the steps described along this work.

References

- ADAMIC, L. A., GLANCE, N. The political blogosphere and the 2004 us election: divided they blog. In *Proceedings of the 3rd international workshop on Link discovery* (2005), ACM, p. 36–43.
- [2] AIROLDI, E. M., BLEI, D. M., FIENBERG, S. E., XING, E. P., JAAKKOLA, T. Mixed membership stochastic block models for relational data with application to protein-protein interactions. In *Proceedings of the international biometrics society* annual meeting (2006), vol. 15.
- [3] ALPAYDIN, E. Introduction to machine learning. MIT press, 2009.
- [4] ANDRADE, G., RAMALHO, G., SANTANA, H., CORRUBLE, V. Extending reinforcement learning to provide dynamic game balancing. In *Proceedings of the Workshop* on Reasoning, Representation, and Learning in Computer Games, 19th International Joint Conference on Artificial Intelligence (IJCAI) (2005), p. 7–12.
- [5] BACKSTROM, L., LESKOVEC, J. Supervised random walks: predicting and recommending links in social networks. In *Proceedings of the fourth ACM international* conference on Web search and data mining (2011), ACM, p. 635–644.
- [6] BAUCKHAGE, C., KERSTING, K., SIFA, R., THURAU, C., DRACHEN, A., CANOSSA, A. How players lose interest in playing a game: An empirical study based on distributions of total playing times. In *Computational Intelligence and Games (CIG), 2012 IEEE conference on* (2012), IEEE, p. 139–146.
- BELKIN, M., NIYOGI, P. Laplacian eigenmaps and spectral techniques for embedding and clustering. In Advances in neural information processing systems (2002), p. 585–591.
- [8] BENGIO, Y., COURVILLE, A., VINCENT, P. Representation learning: A review and new perspectives. *IEEE transactions on pattern analysis and machine intelligence* 35, 8 (2013), 1798–1828.
- [9] BERG, R. V. D., KIPF, T. N., WELLING, M. Graph convolutional matrix completion. In *KDD'18 Deep Learning Day* (London, UK, August 2018).
- [10] BLOCK, F., HODGE, V., HOBSON, S., SEPHTON, N., DEVLIN, S., URSU, M. F., DRACHEN, A., COWLING, P. I. Narrative bytes: Data-driven content production in esports. In *Proceedings of the 2018 ACM International Conference on Interactive Experiences for TV and Online Video* (2018), ACM, p. 29–41.
- [11] BREITKREUTZ, B.-J., STARK, C., REGULY, T., BOUCHER, L., BREITKREUTZ, A., LIVSTONE, M., OUGHTRED, R., LACKNER, D. H., BÄHLER, J., WOOD, V.,
OTHERS. The biogrid interaction database: 2008 update. Nucleic acids research 36, suppl_1 (2007), D637–D640.

- [12] CHAMBERLAIN, B. P., CLOUGH, J., DEISENROTH, M. P. Neural embeddings of graphs in hyperbolic space. arXiv preprint arXiv:1705.10359 (2017).
- [13] CHANG, S., HAN, W., TANG, J., QI, G.-J., AGGARWAL, C. C., HUANG, T. S. Heterogeneous network embedding via deep architectures. In *Proceedings of the 21th* ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (2015), ACM, p. 119–128.
- [14] CHEN, H. Machine learning for information retrieval: neural networks, symbolic learning, and genetic algorithms. *Journal of the American society for Information Science* 46, 3 (1995), 194–216.
- [15] CHO, K., VAN MERRIENBOER, B., GULCEHRE, C., BAHDANAU, D., BOUGARES, F., SCHWENK, H., BENGIO, Y. Learning phrase representations using rnn encoderdecoder for statistical machine translation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)* (2014), p. 1724–1734.
- [16] COURNAPEAU, D. Sci-kit learn. Machine Learning in Python. online, [cit. 8.5. 2017]. URL http://scikit-learn. org (2015).
- [17] DONG, Y., CHAWLA, N. V., SWAMI, A. metapath2vec: Scalable representation learning for heterogeneous networks. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (2017), ACM, p. 135–144.
- [18] EL-NASR, M. S., DRACHEN, A., CANOSSA, A. Game analytics. Springer, 2016.
- [19] ERWIG, M., WALKINGSHAW, E., CHEN, S. An abstract representation of variational graphs. In Proceedings of the 5th International Workshop on Feature-Oriented Software Development (2013), ACM, p. 25–32.
- [20] ESTER, M., KRIEGEL, H.-P., SANDER, J., XU, X., OTHERS. A density-based algorithm for discovering clusters in large spatial databases with noise. In *Kdd* (1996), vol. 96, p. 226–231.
- [21] FIELDS, T., COTTON, B. Social game design: Monetization methods and mechanics. CRC Press, 2011.
- [22] FIGUEIRA, F. M., NASCIMENTO, L., DA SILVA JUNIOR, J., KOHWALTER, T., MURTA, L., CLUA, E. Bing: A framework for dynamic game balancing using provenance. In 2018 17th Brazilian Symposium on Computer Games and Digital Entertainment (SBGames) (2018), IEEE, p. 57–5709.
- [23] FORTUNATO, S. Community detection in graphs. *Physics reports* 486, 3-5 (2010), 75–174.
- [24] FREIRE, M., SERRANO-LAGUNA, Á., IGLESIAS, B. M., MARTÍNEZ-ORTIZ, I., MORENO-GER, P., FERNÁNDEZ-MANJÓN, B. Game learning analytics: learning analytics for serious games. In *Learning, design, and technology*. Springer, 2016, p. 1–29.

- [25] GOODFELLOW, I., BENGIO, Y., COURVILLE, A. Deep Learning. MIT Press, 2016. http://www.deeplearningbook.org.
- [26] GROVER, A., LESKOVEC, J. node2vec: Scalable feature learning for networks. In Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (2016), ACM, p. 855–864.
- [27] GUARDINI, P., MANINETTI, P. Better game experience through game metrics: A rally videogame case study. In *Game Analytics*. Springer, 2013, p. 325–361.
- [28] HAMILTON, W., YING, Z., LESKOVEC, J. Inductive representation learning on large graphs. In Advances in Neural Information Processing Systems (2017), p. 1025–1035.
- [29] HAMILTON, W. L., YING, R., LESKOVEC, J. Representation learning on graphs: Methods and applications. *IEEE Data Engineering Bulletin* (2017).
- [30] HOCHREITER, S., SCHMIDHUBER, J. Long short-term memory. *Neural computation* 9, 8 (1997), 1735–1780.
- [31] HOLMES, G., DONKIN, A., WITTEN, I. H. Weka: A machine learning workbench. In Intelligent Information Systems, 1994. Proceedings of the 1994 Second Australian and New Zealand Conference on (1994), IEEE, p. 357–361.
- [32] KICKMEIER-RUST, M. D. Predicting learning performance in serious games. In *Joint International Conference on Serious Games* (2018), Springer, p. 133–144.
- [33] KIPF, T. N., WELLING, M. Semi-supervised classification with graph convolutional networks. *International Conference on Learning Representations (ICLR)* (2017).
- [34] KOHAVI, R. A study of cross-validation and bootstrap for accuracy estimation and model selection. In *IJCAI International Joint Conference on Artificial Intelligence* (1995), vol. 14, Montreal, Canada, p. 1137–1145.
- [35] KOHWALTER, T., CLUA, E., MURTA, L. Provenance in games. In Brazilian Symposium on Games and Digital Entertainment (SBGAMES) (2012), p. 11.
- [36] KOHWALTER, T., MURTA, L., CLUA, E. Filtering irrelevant sequential data out of game session telemetry though similarity collapses. *Future Generation Computer* Systems 84 (2018), 108–122.
- [37] KOHWALTER, T., OLIVEIRA, T., FREIRE, J., CLUA, E., MURTA, L. Prov viewer: A graph-based visualization tool for interactive exploration of provenance data. In *International Provenance and Annotation Workshop* (2016), Springer, p. 71–82.
- [38] KOHWALTER, T. C., CLUA, E. G., MURTA, L. G. Game flux analysis with provenance. In *Advances in Computer Entertainment*. Springer, 2013, p. 320–331.
- [39] KOHWALTER, T. C., CLUA, E. W. G., MURTA, L. G. P. Reinforcing software engineering learning through provenance. In 2014 Brazilian Symposium on Software Engineering (SBES) (2014), IEEE, p. 131–140.
- [40] KOHWALTER, T. C., DE AZEREDO FIGUEIRA, F. M., DE LIMA SERDEIRO, E. A., DA SILVA JUNIOR, J. R., MURTA, L. G. P., CLUA, E. W. G. Understanding game sessions through provenance. *Entertainment Computing* 27 (2018), 110–127.

- [41] KOHWALTER, T. C., MURTA, L. G. P., CLUA, E. W. G. Capturing game telemetry with provenance. In 2017 16th Brazilian Symposium on Computer Games and Digital Entertainment (SBGames) (2017), IEEE, p. 66–75.
- [42] LI, X., CHEN, H. Recommendation as link prediction: a graph kernel-based machine learning approach. In *Proceedings of the 9th ACM/IEEE-CS joint conference on Digital libraries* (2009), ACM, p. 213–216.
- [43] LI, Y., TARLOW, D., BROCKSCHMIDT, M., ZEMEL, R. Gated graph sequence neural networks. In International Conference on Learning Representations (ICLR) (2016).
- [44] LIBERZON, A., SUBRAMANIAN, A., PINCHBACK, R., THORVALDSDÓTTIR, H., TAMAYO, P., MESIROV, J. P. Molecular signatures database (msigdb) 3.0. *Bioinformatics* 27, 12 (2011), 1739–1740.
- [45] LIU, Y., KOU, Z. Predicting who rated what in large-scale datasets. ACM SIGKDD Explorations Newsletter 9, 2 (2007), 62–65.
- [46] LODHI, H., MUGGLETON, S. Is mutagenesis still challenging. In Proceedings of the 15th International Conference on Inductive Logic Programming, ILP (2005), Citeseer, p. 35–40.
- [47] MAHLMANN, T., DRACHEN, A., TOGELIUS, J., CANOSSA, A., YANNAKAKIS, G. N. Predicting player behavior in tomb raider: Underworld. In *Computational Intelligence and Games (CIG), 2010 IEEE Symposium on* (2010), IEEE, p. 178–185.
- [48] MAHONEY, M. Large text compression benchmark, 2011. http://www. mattmahoney.net/text/text.html.
- [49] MAKEEV, S. Arcade car physics vehicle simulation for unity3d, 2018. https: //github.com/SergeyMakeev/ArcadeCarPhysics.
- [50] MENG, X., BRADLEY, J., YAVUZ, B., SPARKS, E., VENKATARAMAN, S., LIU, D., FREEMAN, J., TSAI, D., AMDE, M., OWEN, S., OTHERS. Mllib: Machine learning in apache spark. *The Journal of Machine Learning Research* 17, 1 (2016), 1235–1241.
- [51] MICHALSKI, R. S., CARBONELL, J. G., MITCHELL, T. M. Machine learning: An Artificial Intelligence Approach. Springer Science & Business Media, 2013.
- [52] MOREAU, L., CLIFFORD, B., FREIRE, J., FUTRELLE, J., GIL, Y., GROTH, P., KWASNIKOWSKA, N., MILES, S., MISSIER, P., MYERS, J., OTHERS. The open provenance model core specification (v1. 1). *Future generation computer systems* 27, 6 (2011), 743–756.
- [53] NASRABADI, N. M. Pattern recognition and machine learning. Journal of electronic imaging 16, 4 (2007), 049901.
- [54] OLSON, D. L., DELEN, D. Advanced data mining techniques. Springer Science & Business Media, 2008.

- [55] OU, M., CUI, P., PEI, J., ZHANG, Z., ZHU, W. Asymmetric transitivity preserving graph embedding. In Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (2016), ACM, p. 1105–1114.
- [56] PEROZZI, B., AL-RFOU, R., SKIENA, S. Deepwalk: Online learning of social representations. In Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (2014), ACM, p. 701–710.
- [57] PHAM, T., TRAN, T., PHUNG, D. Q., VENKATESH, S. Column networks for collective classification. In *Thirty-First AAAI Conference on Artificial Intelligence* (2017), p. 2485–2491.
- [58] RIBEIRO, L. F., SAVERESE, P. H., FIGUEIREDO, D. R. struc2vec: Learning node representations from structural identity. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (2017), ACM, p. 385–394.
- [59] RIVES, A. W., GALITSKI, T. Modular organization of cellular networks. Proceedings of the National Academy of Sciences 100, 3 (2003), 1128–1133.
- [60] ROSVALL, M., BERGSTROM, C. T. Maps of random walks on complex networks reveal community structure. *Proceedings of the National Academy of Sciences 105*, 4 (2008), 1118–1123.
- [61] SCARSELLI, F., GORI, M., TSOI, A. C., HAGENBUCHNER, M., MONFARDINI, G. The graph neural network model. *IEEE Transactions on Neural Networks 20*, 1 (2009), 61–80.
- [62] SCHUBERT, M., DRACHEN, A., MAHLMANN, T. Esports analytics through encounter detection other sports.
- [63] TAMASSIA, M., RAFFE, W., SIFA, R., DRACHEN, A., ZAMBETTA, F., HITCHENS, M. Predicting player churn in destiny: A hidden markov models approach to predicting player departure in a major online game. In 2016 IEEE Conference on Computational Intelligence and Games (CIG) (2016), IEEE, p. 1–8.
- [64] TANG, J., QU, M., WANG, M., ZHANG, M., YAN, J., MEI, Q. Line: Large-scale information network embedding. In *Proceedings of the 24th International Conference* on World Wide Web (2015), International World Wide Web Conferences Steering Committee, p. 1067–1077.
- [65] TOUTANOVA, K., KLEIN, D., MANNING, C. D., SINGER, Y. Feature-rich partof-speech tagging with a cyclic dependency network. In Proceedings of the 2003 Conference of the North American Chapter of the Association for Computational Linguistics on Human Language Technology-Volume 1 (2003), Association for Computational Linguistics, p. 173–180.
- [66] TYCHSEN, A., CANOSSA, A. Defining personas in games using metrics. In Proceedings of the 2008 conference on future play: Research, play, share (2008), ACM, p. 73–80.

- [67] VELIČKOVIĆ, P., CUCURULL, G., CASANOVA, A., ROMERO, A., LIÒ, P., BENGIO, Y. Graph attention networks. *International Conference on Learning Representations* (*ICLR*) (2018).
- [68] VILJANEN, M., AIROLA, A., MAJANOJA, A.-M., HEIKKONEN, J., PAHIKKALA, T. Measuring player retention and monetization using the mean cumulative function. arXiv preprint arXiv:1709.06737 (2017).
- [69] VOLZ, V., RUDOLPH, G., NAUJOKS, B. Demonstrating the feasibility of automatic game balancing. In *Proceedings of the Genetic and Evolutionary Computation Conference 2016* (2016), ACM, p. 269–276.
- [70] WALLNER, G. Play-graph: A methodology and visualization approach for the analysis of gameplay data. In *Foundations of Digital Games* (2013), p. 253–260.
- [71] WANG, D., CUI, P., ZHU, W. Structural deep network embedding. In Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (2016), ACM, p. 1225–1234.
- [72] WEBER, B. G., MATEAS, M. A data mining approach to strategy prediction. In 2009 IEEE Symposium on Computational Intelligence and Games (2009), IEEE, p. 140–147.
- [73] ZACHARY, W. W. An information flow model for conflict and fission in small groups. Journal of anthropological research 33, 4 (1977), 452–473.
- [74] ZAFARANI, R., LIU, H. Social computing data repository at asu, 2009. http: //socialcomputing.asu.edu/.
- [75] ZITNIK, M., LESKOVEC, J. Predicting multicellular function through multi-layer tissue networks. *Bioinformatics* 33, 14 (2017), i190–i198.
- [76] ZOOK, A., HARRISON, B., RIEDL, M. O. Monte-carlo tree search for simulationbased strategy analysis. In *Proceedings of the 10th Conference on the Foundations* of Digital Games (2015).