Entertainment Computing 32 (2019) 100318

Contents lists available at ScienceDirect

Entertainment Computing

journal homepage: www.elsevier.com/locate/entcom

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

Sidney Araujo Melo*, Aline Paes, Esteban Walter Gonzalez Clua, Troy Costa Kohwalter, Leonardo Gresta Paulino Murta

Department of Computer Science, Universidade Federal Fluminense, Niterói, Rio de Janeiro, Brazil

ARTICLE INFO

Keywords:
Provenance graph
Representation learning
Machine learning
Graph embeddings

ABSTRACT

Game Analytics comprises a set of techniques to analyze both the game quality and player behavior. To succeed in Game Analytics, it is essential to identify what is happening in a game (an effect) and track its causes. Thus, game provenance graph tools have been proposed to capture cause-and-effect relationships occurring in a gameplay session to assist the game design process. However, since game provenance data capture 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 game provenance graphs in a latent 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. We evaluate the generalization capacity of PingUMiL when learning from similar games and compare its performance to classical machine learning methods. The experiments conducted on two racing games show that (1) PingUMiL outperforms classical machine learning methods and (2) representation learning can be used to detect long-range cause-and-effect relationships in only partially observed game data provenance graphs.

1. Introduction

Acquiring understandable game metrics is essential to enhance a data-driven game design. This information may be useful for many purposes, such as game balancing [1,2], players behavior understanding [3,4], detection of failures during game design [5,6], or even enhancing in-game monetizing strategies [7,8].

It is necessary to track and remotely gather data from the game sessions to obtain such metrics, a task known as game telemetry [9]. 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 [10]. There are different strategies for gathering and storing data collected from games, ranging from raw logs to structured formats. 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 graph techniques were recently successfully adapted to record game session history while still denoting the elements of the game and the causal relationships between them [11–14].

Game provenance solutions [13] enhance the capture and the building of structured representation of game sessions, subserving further game analysis. This structure includes, for example, the representation of the direct influence between elements, which are depicted through edges connecting sequential nodes in the provenance graph. Still, there is also a need for implementing domain-specific provenance tracking functions, since different games might have different mechanics. Because of that, these domain-specific functions must be implemented by the game developer. One of the needs for domain-specific functions is the *indirect influence*, *i.e.*, a causal relation between non-consecutive events. However, these functions might grow in complexity if developers need to detect long-range indirect influences, that is, if they need to identify the influence between nodes instantiated along distant timestamps 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

* Corresponding author at: Instituto de Computação, Av. Gal. Milton Tavares de Souza, s/n, Niterói, Rio de Janeiro, Brazil
E-mail addresses: sidneymelo@id.uff.br (S.A. Melo), alinepaes@ic.uff.br (A. Paes), esteban@ic.uff.br (E.W.G. Clua), tkohwalter@ic.uff.br (T.C. Kohwalter), leomurta@ic.uff.br (L.G.P. Murta).

https://doi.org/10.1016/j.entcom.2019.100318
Received 25 January 2019; Received in revised form 26 June 2019; Accepted 28 August 2019
Available online 04 October 2019
1875-9521/

and actual movement directions, speed, or assign threat levels regarding all the attributes of the involved entities. Furthermore, there might still exist important influences not foreseen by the game developer, making it difficult to capture and evidence them even in a structured representation such as a provenance graph.

Until now, these difficulties have remained unexplored, and no improvement has been proposed regarding long-range influence detection in game provenance graphs. In this paper, we propose a framework called PingUMiL that uses recent graph-based representation learning techniques [15] to detect long-range influence edges and improve game provenance data capture. Graph representation-learning methods induce a mapping that embeds nodes, edges, or entire (sub) graphs, as points in a low-dimensional vector space [15], which can, therefore, be fed to a downstream machine learning method for tasks such as classification, regression, or clustering [16]. We opted for adopting GraphSAGE (Graph Sample and Aggregate framework) as the graph representation learning method since it is capable of inductively generating node embeddings taking into account the node local neighborhood and its attributes.

By using our framework, the game developer/analyst benefits from automatically detecting edges to complete provenance graphs in two scenarios: (i) within a game that lacks some edges, for example, in case of graphs captured with older provenance capture functions or with older versions of the game, or (ii) the game designer or developer has not realized influences which, in their turn, were not captured during game sessions. Even though in this work, we focus on automatically finding long-range influence edges, detection of other types of edges may also be accomplished by the framework with minor modifications.

First, we define the PingUMiL framework, presenting all steps necessary for combining the provenance graph data structure, a graph-based representation learning method, and a downstream machine learning method. Then, we evaluate the benefits of our graph-based framework by comparing its performance in terms of quantitative metrics (precision, recall, f1 score) against feeding classical machine learning methods with raw log data (without provenance graph structure). The gameplay session data for the case studies were captured from the provenance graphs of two racing games. In summary, we aim at answering two main questions:

- Q1: Is it possible to detect long-range influences among the components of a game using machine learning?
- Q2: Does provenance graph representation learning improve the detection of influence edges compared to classical machine learning techniques without provenance?

The experiments presented in this paper point out that long-range influence detection is not only possible but better performed when graph representation learning is employed.

The remaining of the paper is organized as follows: Section 2 presents a guiding example to contextualize both the background and our framework proposal sections. Section 3 presents some recent work in Game Analytics and Machine Learning and background about Provenance in Games and Machine Learning on graphs. Section 4 presents our framework proposal. Section 5 shows two case studies about racing games, explaining in detail how the framework was applied to them. Section 6 presents, analyzes, and discusses our experimental results. Finally, Section 7 concludes this work, pointing out future works.

2. Guiding example

Let us consider a game development studio named *Radical Games* that employs provenance graphs for tracking game data in their racing

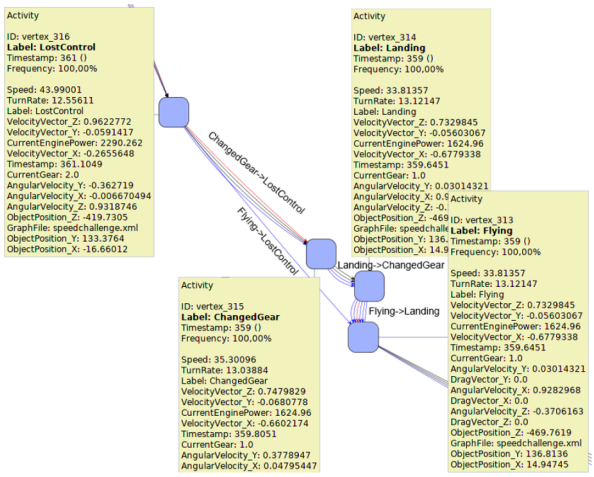


Fig. 1. Subgraph from *Speed Challenge* game.

game *Speed Challenge*. For this racing game, the player nodes are captured and contain the following attributes: player position (3d vector), speed, current gear, current engine power, turn rate, velocity (3d vector), angular velocity (3d vector), and a provenance label. Provenance labels are tags for the current agent, activity, or entity node. Possible values for the provenance label are:

- Player ID, for agent nodes.
- Actions from the list ChangedGear, Brake, Crash, Flying, Landing, LostControl, Scraped, for activity nodes.

Since *Speed Challenge* is a simple racing game, no entity nodes are instantiated. The game has no items, collectibles, power ups or interactive obstacles. Example of nodes are shown in Fig. 1.

According to the game design guidelines, game programmers from *Radical Games* must implement domain-specific methods to collect nodes and connect them to express influence relationships using a Provenance tool. One of the desired influence relationships specified by the game design guidelines is “After driving fast over a slope, a player’s car loses contact with the ground (Flying event), and the player loses control of the vehicle (LostControl event) due to its increasing speed.” The “Flying” event is detected through the game engine provided collision methods. The “LostControl” event is detected using a physics formula whose inputs are velocity and angular velocity vectors, and there is also a constraint regarding the increase of the speed attribute. Activity nodes for all the actions/events are generated by the employed Provenance tool, and if these nodes are instantiated respecting influence triggers defined by game programmers, edges are created between them, materializing the influence relationship. An example of the relationship mentioned above is depicted in Fig. 1, where a Flying Node (index 313) is connected to LostControl Node (index 316). Events unrelated to that relationship can happen between the Flying and LostControl nodes, such as the Landing node (index 314). Influence relationships such as “Every Landing event is a consequence of a previous Flying event” and “if the player has Lost Control of the vehicle and Crashed less than 2 s after, player nodes representing these events are connected” are similarly captured.

After implementing all these functions, it is possible to capture a subgraph such as the one presented in Fig. 2, where Player A (red nodes) and Player B (blue nodes) race each other on a section of the track.

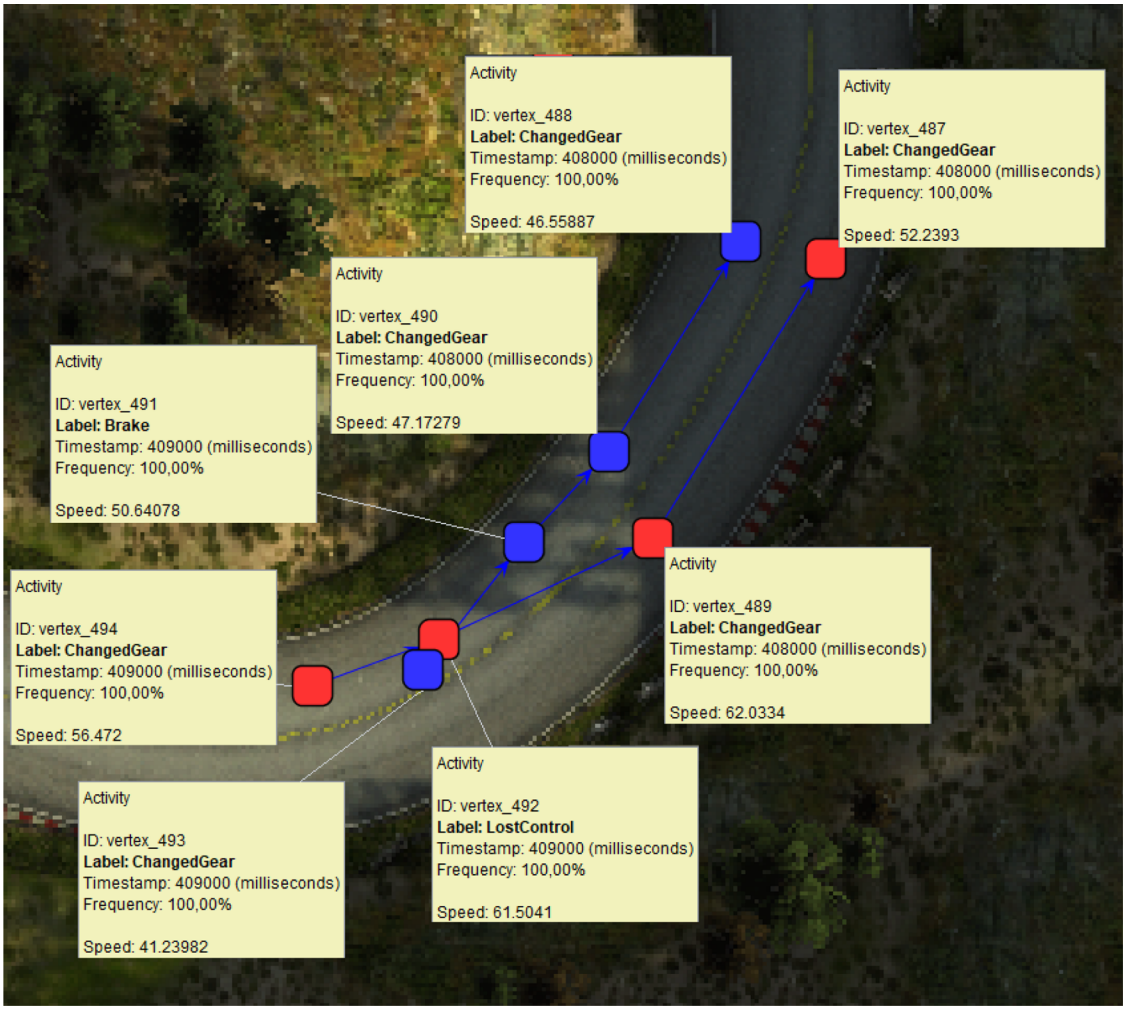


Fig. 2. Subgraph from *Speed Challenge* in a multiplayer race.

3. Background

This Section presents related works and two fundamental topics for understanding PingUMiL: Provenance in games and Learning graphs embeddings. We start by contextualizing this work with other works combining Machine Learning and Game Analytics. Then, we define provenance in the game context, followed by a review on previous game provenance works and a discussion on the weak and strong points in the game provenance capture process. Next, we overview recent advances in graph representation learning and present key concepts for our framework, such as node embedding and neighborhood aggregation.

3.1. Related work

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 the procedural generation of content, prediction (recommendation systems, event prediction), and game learning analytics [17,18].

Schubert et al. [17] present a technique for segmenting matches of the Multiplayer Online Battle Arena (MOBA) game DOTA into spatio-temporally defined components called encounters. Through 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.

Block et al. [18] present 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 performances metrics of each player to thousand of historical data, calculating how many percents of historic performances are exceeded in the game session.

Freire et al. [19] combine 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-Rust [20] combined learning performance metrics and log files from serious games to predict learning performances in serious games.

The works mentioned above 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, our work is an attempt to support and improve a game analytics process (in our case, Provenance in Games [11]) using machine learning regarding causal relationships between game components. To the best of our knowledge, there are no works directly related to the framework proposed in this paper.

3.2. Provenance in games

The adoption of data provenance in the context of games was first proposed by Kohwalter et al. [11] through the PinG (Provenance in Games) framework. Following the Open Provenance Model [21], the authors defined 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 NPCs (Non-playable characters) 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 are mapped as *Activity* nodes. Causal relationships between game elements are mapped as edges connecting their respective nodes, resulting in a game data provenance graph. Causality indicates a relationship between two events where one event is affected by the other. The provenance approach materializes causal relationships explicitly defined by the game developer. Therefore, each edge captured through a provenance approach represents the consequences between the game objects' actions and states.

The PinG framework was implemented in [22] as a domain-dependant prototype and, later, in [13] 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 smoother provenance capture, requiring minimal coding in the game's existing components [14]. Besides the PinGU plugin, graph-based visualization tools such as Prov Viewer [23] have also been developed for post-game session analysis.

Provenance capture through PinGU can be easily employed in a game by following four steps: (1) instantiate and attach core provenance scripts, (2) identify actions and interactions, (3) implement domain-specific provenance capture functions, and (4) export provenance data. These domain-specific provenance tracking functions must be attached to game elements scripts in a four-step recipe: (3.1) add game-related attributes, (3.2) create nodes, (3.3) check for influences affecting the current event or state, and (3.4) generate influence triggers for future events or states. A simple example is the "Every Landing event is a consequence of a previous Flying event" influence. It takes two functions: one that is called whenever the car loses contact with the ground, inserting a Flying node in the graph and generating an influence trigger for the next Landing node, and another that is called whenever the car comes in contact with the ground, inserting a Landing node and searching for a previous influence trigger.

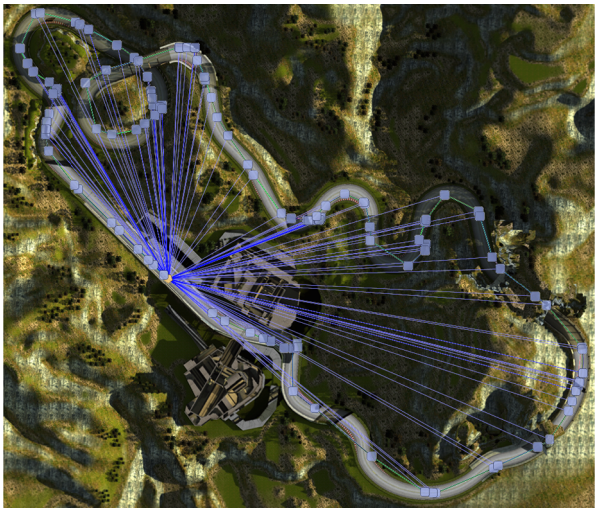


Fig. 3. Racing game provenance graph example.

One of the most prominent advantages of provenance graphs is its richness of detail, i.e., the data collected at fine-grain Figs. 1–3 present examples of racing game provenance graphs and their 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 crucial to notice that a single lap has generated 140 nodes and more than 700 edges in less than a two-minute gameplay session.

In Fig. 1, it is possible to observe all the attributes of a car controlled by the player during the *Flying*, *Landing*, *ChangedGear*, and *LostControl* activities. While PinGU implements several methods to facilitate provenance capture, game developers must write domain-specific provenance tracking functions and attach them to each entity in the game [14]. 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 capture algorithms and the events happening within a game session influence directly the amount of generated data [14]. Also, this rich and raw provenance data can be used in machine learning tasks, to describe hidden patterns, aid the game maintenance, and help in future developments. These are the main assumptions of this research.

Fig. 1 also presents activity nodes connected by edges, which represent the causal relationship between these activities. Edges, in their turn, are capture according to their influence range. Direct edges are edges connecting sequential nodes. Most direct edges are automatically capture by PinGU due to their temporal relationship. In this context, PinGU assumes that the current node (and its attributes) of a game object is temporally related to its previous node. Influence edges, similarly to node attributes capture, are capture through domain-specific provenance tracking functions and can connect any pair of nodes in order to model a cause and effect relationship of interest. Therefore, influence edges can represent cause and effect relationships. We focus on indirect influences. These indirect influences are provided by the game designer, which, in turn, defines guidelines for methods that will capture these influences.

For example, Fig. 1 shows a subgraph where the relationship "After driving fast over a slope, a player's car loses contact with the ground (Flying event) and the player loses control of the vehicle (LostControl event) due to its increasing speed" occurs and is captured. It is possible to observe that nodes 314 (Landing event) and 315 (ChangedGear event) are omitted from the relationship mentioned above. Furthermore, node 316 (LostControl event) is sequentially connected to node 315 (ChangedGear event) due to their temporal relation and because they belong to the same agent (Player).

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, a considerable amount of effort has also to be put by the game developer and the game designer when implementing or adapting domain-specific provenance capture algorithm, 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 detect long-range influences as a first attempt to enhance and automate game provenance data capture.

3.3. Machine learning on graphs

Recently, efforts to learn representations from graph-structured data with machine learning tasks have gained attention [24–29]. Most machine learning methods in this area seek to make predictions, discover new patterns, or classify nodes by transforming graph-structured data into feature information [30,31,15]. To that, these graph representation learning methods incorporate information about the structure of the graph to yield latent features that represent some properties of the data [25,32,33,29]. 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 [15]. Vector representations resulting from mapping nodes into the learned space are called node embeddings.

Hamilton et al. [15] point out that node embedding methods rely on four components:

- A pairwise proximity function, which measures how closely connected two 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 [34], HOPE [26], DeepWalk [25], node2vec [27] generate vector representations for each node independently. These methods rely on matrix factorization approaches for dimensionality reduction [34] (Laplacian Eigenmaps, HOPE) or random walk statistics (DeepWalk, node2vec). However, they do not consider node attributes during the encoding. This is a major drawback since node attributes can be highly relevant for representing a node.

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 values of the node's features. At each iteration, the node's embeddings aggregate their neighbors' embeddings, generating new embeddings. These approaches determine the node embedding according to its surrounding neighborhood attributes. Therefore, they are also called neighborhood aggregation methods. Examples of these methods are Graph Convolutional Networks (GCN) [33], Column Networks [35], and GraphSAGE [29]. Still, they 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. Extensions and solutions have been lately proposed for handling these limitations [36–38].

In this research, we opted for GraphSAGE [29] (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 sample neighborhood.

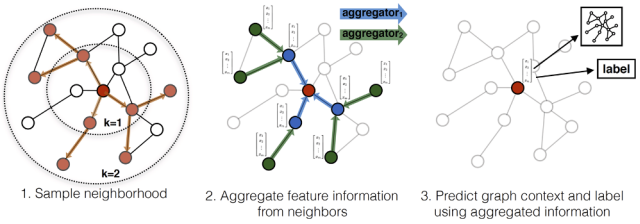


Fig. 4. Visual illustration of the GraphSAGE sample and aggregate approach [29].
Source: http://snap.stanford.edu/graphsage.

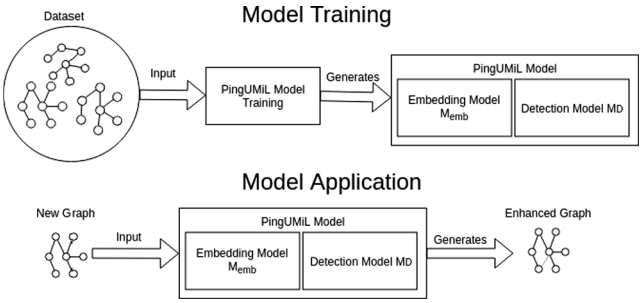


Fig. 5. Overview of the PingUMiL framework.

Fig. 4 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 function to fine-tune weight matrices and aggregation functions' parameters. This graph-based loss function enforces similarities on representations of nearby nodes.

4. PingUMiL

In this section, we present our PingUMiL[1] approach, which is a framework for using Machine Learning techniques to add missing influences in the captured game provenance using a graph-based representation through Machine Learning. The idea behind PingUMiL is to create machine learning models capable of, given learned vector representation of pair of nodes, determine if a type of influence occurs between these pair of nodes. Fig. 5 presents an overview of the PingUMiL framework.

However, before going deeper into the framework's workflow, it is vital to define Machine Learning terminology in the current paper's context. The framework encompasses two phases: model training and model usage.

During model training, Machine Learning algorithms receive as input a dataset related to the task to be learned. A data point in a dataset is called an *example*. In a supervised setting, an example is composed of a set of feature attributes $X = [X_1, X_2, ..., X_k]$ and a *target* attribute y. In a *classification* task, the target is within a discrete set of values. Thus, an example can be seen as a vector $[X, y]$. In our case, the attributes of an example are the data associated to a pair of nodes, and we have a classification task where the target is a Boolean value indicating the existence or not of an edge connecting those nodes.

Here, the *training* phase encompasses the act of feeding the set of positive (target y value is True) and negative (target y value is False) *examples* to an algorithm that tries to learn a model M capable of classifying data accordingly, based on the provided data. In the model usage phase, we call *prediction* the process of feeding new (real-world) data (a set of attributes) to the trained model M to *classify* the *target* y accordingly; in our case, real-world data is composed of the attributes

[1] https://github.com/sidneyaraujomelo/PingUMiL.

related to a pair of nodes. Thus, given a *candidate* edge, the trained model will try to *predict* if the candidate edge should be taken as a valid edge and inserted into the enhanced graph, or taken as an invalid edge and ignored. Finally, we call *detection* the process of, given a new graph, sampling *candidate edges* and *predicting* their targets.

The general procedure of the framework proposed in this work 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 candidate edge is a valid edge. For example, we may intend to know whether or not a given edge candidate is a long-range influence edge.

We divided this section into three subsections. Before detailing the steps, we introduce a motivation scenario in Section 4.1 based on our guiding example that is revisited in each of the following subsections. Then, we divided the proposed framework for detecting edges in two steps: training (4.2) and usage (4.3).

4.1. Motivation scenario

Consider the guiding example presented in Section 2. There are some influence relationships whose existence are harder to determine using formulas and functions that leverage only node attributes but are easily perceived by observation. For instance, Bob, who is a Game Analytics expert from Game Studio Radical Games, analyzes the subgraph from Fig. 2 and notices that Player B (blue nodes) has braked in node 491 due to Player A (red nodes) overtake and control loss to avoid a collision. If they had collided, a crash node would have been instantiated in the graph and connected to both Player As control loss and Player Bs break. However, relationships such as "Player A control loss influences Player B braking" are not trivial to determine through a function or formula. The loss control event from A, the brake event from B, and several attributes such as players positions in the track, angles, distance, and speed must be taken into account to determine if an edge should exist between these events. Otherwise, the function might be more inclined to determine edges that do not represent the intended relationship but are taken as representative by the function (here called as false positive edges). Also, the number of interacting agents during gameplay adds more variables to the problem.

On the other hand, Bob, as a Game Analytics expert, can perceive more easily when the relationship mentioned above occurs by observation. By analyzing a recording of the gameplay session, which could be a provenance graph (such as Fig. 2) or a gameplay video, Bob can visually determine whenever "Player A control loss influences Player B braking" occurs, add edges manually to the analyzed provenance graph, and obtain a more detailed representation of the gameplay session. However, the higher the number of graphs, the higher the cost of introducing these new edges.

In this context, Bob would benefit from a solution that, given the game provenance graphs and his example annotation of desired edges, could generate a model capable of automatically detecting and adding these edges in other similar game provenance graphs. In a real-life scenario, Bob would annotate graphs with the desired edges and feed them to our Machine Learning solution, which would, in its turn, generate a model for detecting the desired edges. Afterward, Bob can use the model on every new graph from *Speed Challenge* to automatize the desired edges detection.

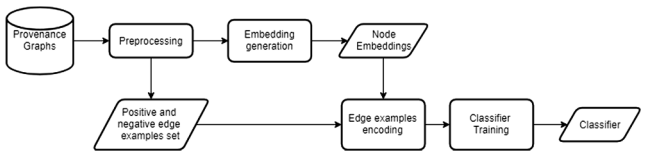


Fig. 6. Overview of the proposed framework.

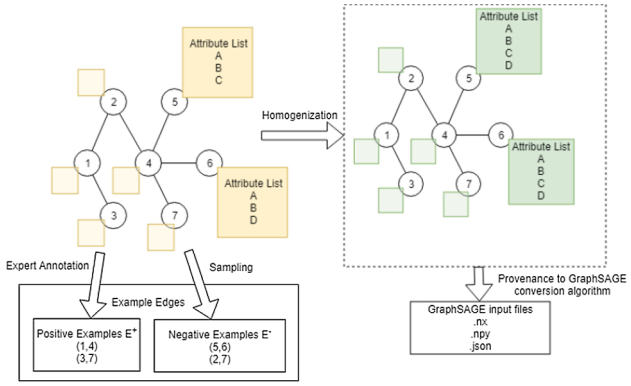


Fig. 7. Overview of the preprocessing step.

4.2. Model training

An overview of PingUMiL's model training is shown in Fig. 6. A set of provenance graphs is the input to the whole framework. These graphs must be preprocessed due to node heterogeneity and the definition of balanced sets of positive and negative example edges for the downstream classification tasks. Positive example edges are edges that represent an influence relation between their connecting nodes, while negative example edges are edges that do not represent an influence relation between their connecting nodes. 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.

4.2.1. Pre-processing

The preprocessing step is fundamental for both the node/edges embedding generation and the training of the classifier tasks and encompasses two main tasks: acquiring positive and negative examples and homogenizing graphs. Fig. 7 presents an overview of the preprocessing step, where each circle represent a node and each box a list of attributes attached to that node. Orange boxes represents original lists of attributes, which might differ from node to node in an heterogeneous graph, and green boxes represents lists of attributes after homogenization. Given an input graph, positive example edges set is defined by an expert's annotation, and negative example edges are sampled from the input graph. Also, the graph is homogenized in order to provide a specific data structure for subsequent embedding generation step. Optionally, the positive example edges and their connecting node's attributes are used to define the sample criterion for sampling negative example edges. For both tasks, it is essential to formally define the provenance graphs in order to understand the algorithms developed for this step.

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 an attribute $a \in A$ 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 attributes so that each attribute 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 [29], 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 $t' = \{l | l \in t_i, i = 1, ..., |T_v|\}$. 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 attributes' order and default values for previously unaddressed features. We illustrate that in Fig. 7, where the input graph contains 7 nodes of two types: (A,B,C) and (A,B,D). The output graph, on the other hand, contains only 1 type of node: (A,B,C,D). In this case, for example, the homogeneous node 5 receives all values from its heterogeneous version and a default value for the attribute D. In a real-world scenario, the default value for an attribute should be determined by the game designer or analyst. For example, a possible default value for the *Speed* attribute is 0 if a game object does not possess a *Speed* attribute, assuming that it probably does not move.

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.

Positive and negative example edges are necessary for following binary classification tasks. Positive examples, as shown in Fig. 7, can be obtained through expert annotation. In our example scenario, Bob is interested in the "Player A control loss influences Player B breaking" influence. In this step, Bob would make annotations on available graphs in order to generate the set of positive example edges. After annotating positive example edges, Bob could randomly sample disconnected pairs of nodes to compose the set of negative example edges. However, in order to enhance the quality of the training, he could also apply some criteria based on his set of positive example edges to generate a more competitive set of negative example edges.

Positive example edges types are used to determine a sample criterion for negative example edges. Let $L_v = l_1, ..., l_n$ be the set of node's provenance label values, where l_i is a provenance label such as "Flying", "ChangedGear" or "Landing". Then, we define edge types as $z_e = l_i \rightarrow l_j$ where $l_i, l_j \in L_v$. For example, an edge connecting a node with the "Flying" provenance label with another node with the "Landing" provenance label has the type "Flying → Landing". We define Z_{E^+} as the set of edge types of all positive example edges. This defines the first criterion for acquiring negative example edges, i.e., a negative example edge e^- must have an edge type $z_k \in Z_{E^+}$. Therefore, the set of negative example edges types is defined as $Z_{E^-} \subseteq Z_{E^+}$.

Consider that, in our guiding example, Bob determines target influence edges such as those between nodes 491 (Brake event) and 489 (ChangedGear event), shown in Fig. 2, as positive example edge, i.e., an edge with edge label ChangedGear → Brake and apply the aforementioned criteria, i.e., $L_{E^+} = (ChangedGear \rightarrow Brake)$. Consequently, any negative edge should have the label ChangedGear → Brake.

Still in our guiding example, Bob could check the range of values of attributes in the positive edge examples and use its values as a criterion to enhance the quality of the negative edge examples. For example, in Fig. 2, our positive example is composed of a Brake event with timestamp 409 and speed 50 and the ChangedGear event with timestamp 408 and speed 62. A simple criterion could prune negative examples with speed values much above or much below these speed values. Another criterion could prune negative examples whose timestamp differences are much larger than 1. These criteria are optional and could both be determined by the analyst or derived from game design rules.

In our example scenario, Bob could use scripts already implemented for negative sampling example edges using edge types and timestamp difference as criteria.

Algorithm 1. Provenance to GraphSAGE files conversion algorithm

```
input : Graph G(V, E), function
        defineSet :→ {"train", "test", "val"}
output: NetworkX Graph nx; Id Map id_map;
        Class Map c_map; features array f
nx = {};
id_map = {};
c_map = {};
f = [[]];
for v ∈ V do
    original_id = GetID(v);
    id = length(id_map);
    id_map[id] = original_id;
    nx.add_node(id);
    nx[id]["set"] = defineSet();
    for every element el ∈ v and el is not the ID
     element do
        f[id].append(el.value);
        if el is the class element then
            c_map[id] = el.value;
for e ∈ E do
    original_source, original_target =
     getNodesOriginalID(e);
    source = getMapKey(id_map, original_source);
    target = getMapKey(id_map, original_target);
    nx.addEdge(source, target);
    nx[source,target]["set"] = defineSet();
```

GraphSAGE is fed with graphs in the NetworkX[2] 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. 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 their connecting nodes ids and pairs of attribute-values. A tree-traversing based algorithm could be used to visit provenance nodes and store the id, attributes, and class values to their respective files, 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 supervised setting, as well as edges for the unsupervised setting.

The pseudocode described in Algorithm 1 covers most requirements

[2] https://networkx.github.io/.

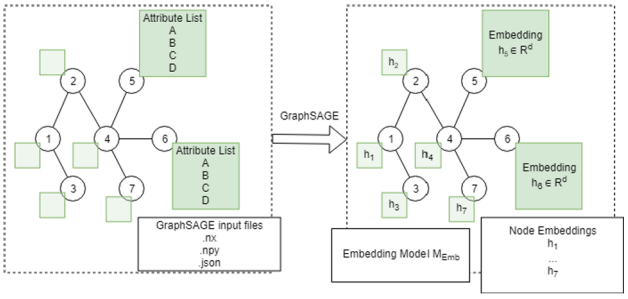


Fig. 8. Overview of the embedding generation step.

to generate GraphSAGE required files from a provenance graph. The algorithm traverses through node elements and distribute their ids, classes, and attributes into separate data structures. Then, it traverses through edge elements, adding edges to the NetworkX graph instance. The function *defineSet* distributes edges and nodes into train, test, and validation sets. Also, the algorithm should take into account the order of the attributes, since the heterogeneous nodes might have different attributes, and generate homogeneous nodes that have all attributes in the same order, including default values for unknown attributes. This is not covered in Algorithm 1 for conciseness sake.

Back to our illustrative scenario, after annotating similar positive example edges that represent the influence edge Player A control loss influences Player B braking, such as edge (491,489) in Fig. 2, and sampling negative example edges, Bob uses a script from PingUMiLs framework to homogenize and convert the *Speed Challenge* graphs to GraphSAGEs required input files.

4.2.2. Embedding generation

As aforementioned, embedding generation is done using the GraphSAGE framework, which is based on neighborhood aggregation techniques and includes several aggregation methods based on functions and neural network architectures. These neighborhood aggregation techniques are used to aggregate neighbor nodes' attributes to generate node embeddings that leverage both the node and its local context. An exhausting list of aggregation methods implemented within GraphSAGE is:

1. Mean-based aggregator: this architecture generates embeddings by taking the element-wise mean of feature vectors.
2. LSTM-based aggregator: this architecture generates embeddings by feeding a random permutation of the 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.
4. 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).

Fig. 8 presents an overview of this step. 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. Also, these embeddings can be used for other downstream machine learning tasks such as clustering and regression, for example. Also, GraphSAGE generates a trained model for embedding generation M_{Emb}, whose function is to generate embeddings for new graphs using parameters and weights learned for older graphs of the same domain. We refer to models M_{Emb} generated by GraphSAGE as embedding models from now on.

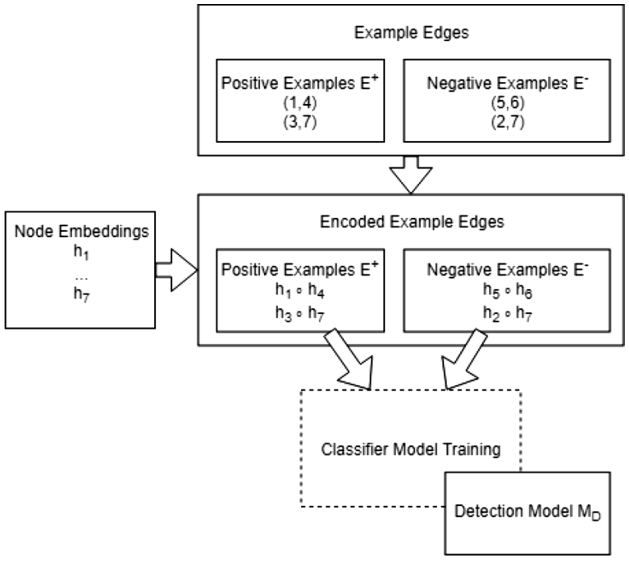


Fig. 9. Overview of the classifier training step.

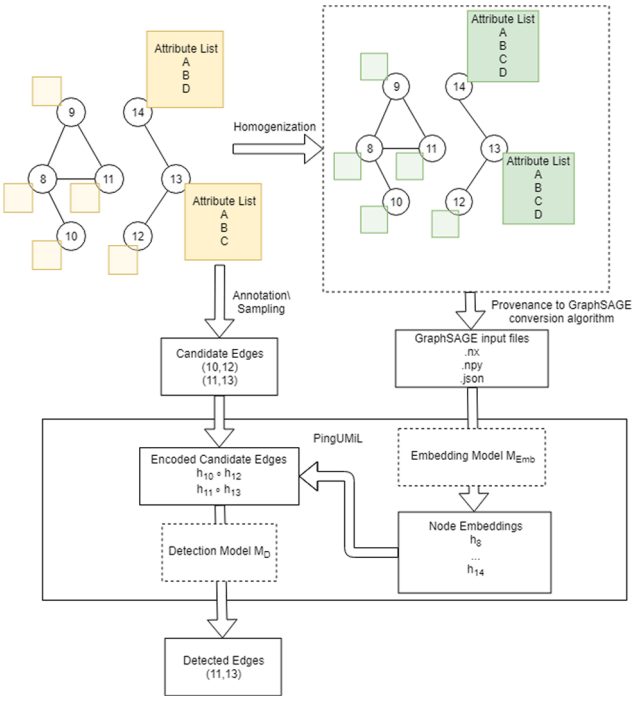


Fig. 10. Overview of the model usage.

Back to our scenario, Bob generates node embeddings h_v for all nodes v from all graphs he has used for annotation using a GraphSAGE aggregation technique of his choice and an embedding model M_{Emb}.

4.2.3. Classifier training

Classification tasks can be done 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 several different libraries and packages available that implement machine learning algorithms, such as Apache Spark's MLLib [39], SciKit-Learn [40], and WEKA [41]. Any of these classification tools require input examples in the form of feature vectors and their respective target classes, 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 space. Since generated node embeddings are vectors, all with the same dimension, it is possible to generate edge embeddings using a simple function $embed\_edge(v_1, v_2) = h_{v_1} \circ h_{v_2}$ where v_1 and v_2 are the connecting nodes of edge e and $\circ$ is an operation such as concatenation, element-wise sum or cross product. Positive examples, representing long-range influence, receive target class 1, while negative examples, which are long-range edges that do not represent an influence relation between their connecting nodes, receive target class 0.

Finally, classifier training 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. Both edge encoding and classification training are illustrated in Fig. 9.

In our scenario, Bob uses PingUMiL's classifier training script to generate the detection model M_D. In this script, he inputs node embeddings, sets of positive and negative examples, chooses an encoding function between the ones available (concatenation or elementwise multiplication) and SciKit-Learn's classifier model (such as MLP, SGD). At the end of the process, the PingUMiL's script outputs a detection model M_D trained to detect the desired type of edge. Now that Bob has an embedding model M_{Emb} and a detection model M_D for an influence edge of his interest, he can apply them into new graphs to automatically detect such edges.

4.3. Model usage

The model usage phase is illustrated in Fig. 10 and is very similar to the model training phase. The resulting model M_D is trained to detect encoded candidate edges. Given a new graph G_n that should be enhanced using model M_D, all its nodes must be embedded using the same architecture and settings employed in the Embedding generation step, i.e., homogenized and embedded using the embedding model M_{Emb} produced in step Embedding Generation. Parallel, candidate edges should be annotated or sampled to reduce the search space and encoded using the same encoding function employed in the Classifier training step. Encoded edges are fed to the trained model M_D in order to be predicted as a target edge or not. If positively predicted, candidate edges are added to the graph. Still, strategies for sampling candidate edges from this new graph are also needed. Otherwise, one should 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, as 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 capture, similar to a search window approach, excluding the ones that are already connected.
- Another approach would be to set edge type constraints such as those defined in the preprocessing step, i.e., candidate edges type must belong to the target edge type's set. Both search window and edge type constraint approaches could be applied simultaneously.

It is worth mentioning that by not using the edge type constraints approach, the trained model might predict candidate edges positively with edge types differently from the ones defined in positive examples training phase. These different edge types come from nodes whose labels were not used in the training step. That means, it is possible that the model M_D detects an influence between pairs of nodes with label sequence unobserved in the model training. For example, consider that Bob used only edges of type Brake → ChangedGear in the model training and experimentally opted not to use a type constraint approach. It is possible that an edge Brake → Flying is sampled as a candidate edge and positively classified as a valid influence by model M_D. We name this possible phenomenon as "influence discovery." The aggregation approach and deep learning architectures used in Embedding generation step are the premises for the validity of discovered influences.

The aggregation methods aggregate information from sampled neighbors. In other words, in these methods, a nodes neighborhood affects its resulting embedding. Also, it is well known that deep learning techniques are capable of learning relevant unforeseen features and structures from their inputs. Therefore, candidate edges with unobserved provenance edge types (in our previous example, Brake → Flying candidate edge), classified positively by the resulting model M_D, could also represent existing influences between nodes, similar to the ones used in the training step (in our previous example, Brake → ChangedGear example edges). Evaluation of these kinds of candidate edges could be performed by game analytics and game design experts. Also, we intend to investigate the use of this frameworks resulting models on influence discovery soon.

In our final scenario example step, Bob makes a new *Speed Challenge*'s graph G_k homogeneous. After that, Bob uses the edge type constraint approaches and samples several candidate edges of type Brake → ChangedGear across graph G_k. Finally, he uses a PingUMiL script that takes the homogeneous graph G_k, the sampled candidate edges, the embedding and detection models M_{Emb} and M_D as inputs and outputs an enhanced graph G_k^+ with automatically detected Brake → ChangedGear edges. Also, future graphs can be processed by the influence edge detector, so that Bob can resume his game analytic tasks with more thoroughgoing working material.

5. Case study

The following sub-sections illustrate our proposal applied in two racing game prototypes, detailing every step of the framework. 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),[3] presented in Fig. 11a, and the second one is Arcade Car Physics (AC),[4] presented in Fig. 11b. 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 car physics. AC implements several algorithms over or instead native engine physics, such as Speed Curve, Ackermann Steering, and Stabilizer Bar Forces [42].

Beyond serving as examples for the PingUMiL framework application, both case studies are used for answering the research questions stated in the introduction. That is, all graphs, edge sets, and models produced by the steps described in the following subsections are used in our Experimental Results Section (6).

All datasets and codes mentioned in this section can be found in the framework's Github repository.[5]

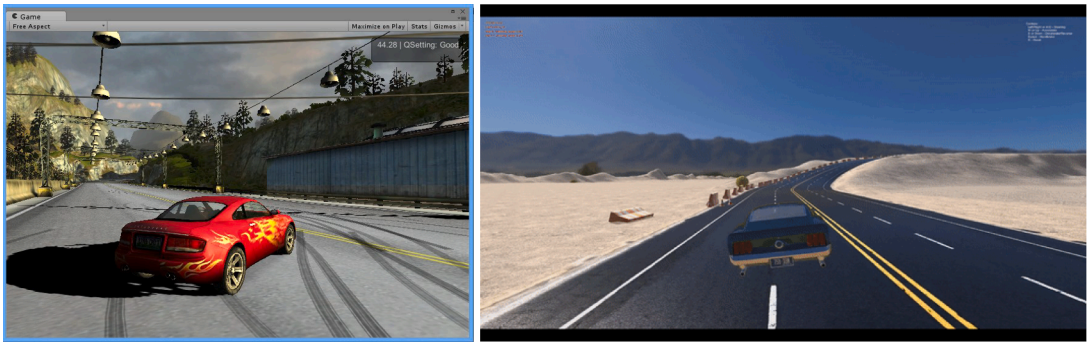
5.1. Graph capture

First, we captured provenance graphs from game sessions. We used the provenance capture algorithm for CT developed by Kohwalter et al. [14] in CT and adapted with minor changes for AC, mostly due to differences in physics implementation between both games. The provenance capture algorithm for CT already implemented a simple long-

[3] https://assetstore.unity.com/packages/templates/tutorials/car-tutorial-unity-3-x-only-10.

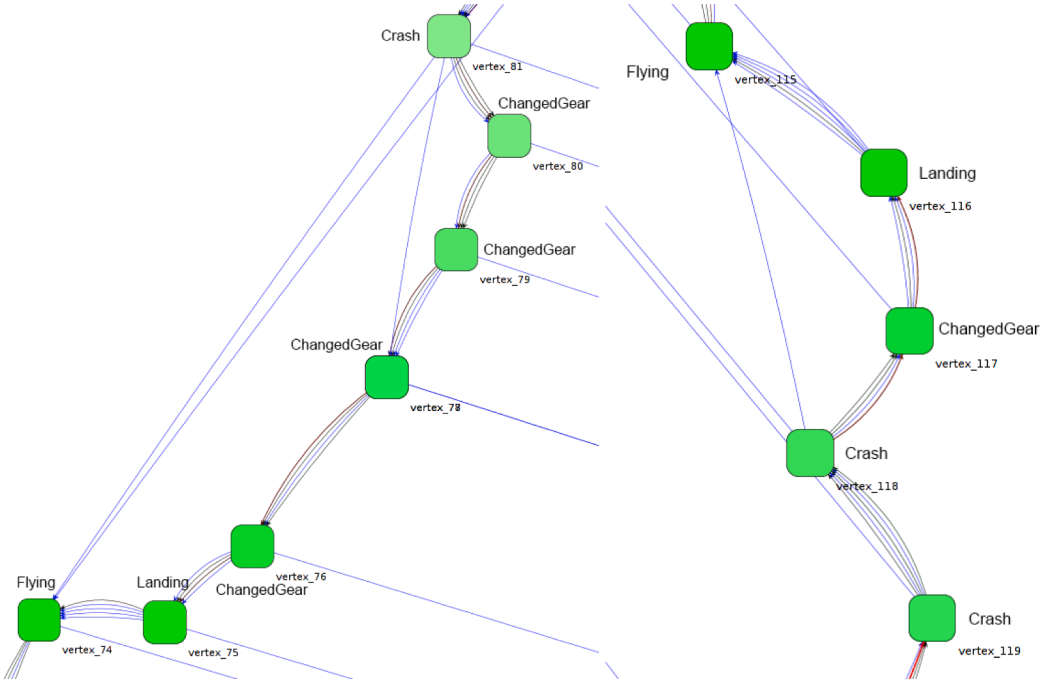
[4] https://github.com/SergeyMakeev/ArcadeCarPhysics.

[5] https://github.com/sidneyaraujomelo/PingUMiL.



(a) Car Tutorial Unity screenshot. (b) Car Arcade Physics screenshot.

Fig. 11. Racing games screenshots. (a) Car Tutorial Unity screenshot. Source: https://answers:unity:com = questions = 582986. (b) Car Arcade Physics screenshot.



(a) Positive example. (b) Negative example.

Fig. 12. Flying → Crash edge examples.

range influence edges capturing the following type of edges: Crash → Crash, Crash → LostControl, Crash → Scraped, Flying → Crash, Flying → Landing, Flying → LostControl, Flying → Scraped, HandBrake → LostControl, Scraped → Crash, and Scraped → Scraped.

Most of these influences have been implemented using a time-constrained function. For example, the Crash → LostControl edge materializes the influence A Control Loss event is a consequence of a Crash event that happened at most 2 s earlier. In our experiment, we consider these edges as experts annotation and, therefore, as positive examples. However, it is not our intent to question the validity of these edges or their capture functions.

For CT, 10 game session graphs were capture, which in total contain 9194 nodes and 47,497 edges. For AC, 3 game session graphs were capture, which in total contain 4146 nodes and 21,397 edges. Game sessions were recorded by a group of three testers with no link to this research. The testers were all male, with age between 20 and 35 years and considered themselves experienced players.

5.2. Preprocessing

Provenance capture algorithm for CT [14] already implemented long-range influence edges capture for a previously enumerated type of edges. As aforementioned, the same provenance algorithm for CT was adapted and implemented in AC, i.e., the same type of influence edges was also captured for AC.

Since we take these long-range influence edges as positive examples annotated by experts, we need to remove them from the graph in this step for both CT and AC. This is necessary for simulating a scenario where the input graph does not contain the target edges.

Table 1
Total number of positive and negative examples of CT.

Edge Type	Positive Examples	Negative Examples
Crash → Crash	98	138
Crash → LostControl	16	230
Crash → Scraped	3	150
Flying → Crash	199	266
Flying → Landing	42	507
Flying → LostControl	90	343
Flying → Scraped	1	265
HandBrake → LostControl	62	68
Scraped → Crash	81	78
Scraped → Scraped	2	99
Total	594	2144

Table 2
Total number of positive and negative examples of AC.

Edge Type	Positive Examples	Negative Examples
Crash → Crash	38	31
Crash → LostControl	11	85
Crash → Scraped	6	127
Flying → Crash	51	125
Flying → Landing	136	278
Flying → LostControl	29	165
Flying → Scraped	8	217
HandBrake → LostControl	4	15
Scraped → Crash	56	38
Scraped → Scraped	11	153
Total	350	1234

Table 3
Total number of edge examples of both CT and AC.

Edge Type	CT Examples	AC Examples
Crash → Crash	196	62
Crash → LostControl	32	22
Crash → Scraped	6	12
Flying → Crash	398	102
Flying → Landing	84	272
Flying → LostControl	180	58
Flying → Scraped	2	16
HandBrake → LostControl	124	8
Scraped → Crash	156	76
Scraped → Scraped	4	22
Total	1182	650

Once the positive example set is defined, it is possible to define the sample criterion for sampling negative edge examples. The first sample criterion is the set of types of positive examples. In this case, negative examples would necessarily connect two nodes with provenance labels represented in the positive example set, respecting the node label's order. For better understanding, a positive and negative example of Flying → Crash edges are shown in Fig. 12. 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 exhibited in Fig. 12a presents an influence edge of type Flying → Crash 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, (s) he is not able to prevent the crash. A case of a candidate negative example is shown in Fig. 12b. 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 199 is already a direct consequence from Crash node 118.

The second criterion is derived from the provenance capture algorithm itself. Kohwalter et al. [14] define a time constraint of two seconds for non-consecutive sequential nodes with most label sequences defined in the first criterion. We define a relaxed constraint based on node path distances by using a search window of 10 nodes. That is, given a node v, we check every non-adjacent node u whose path distance to v is smaller than 10 and check if the type of the virtual edge $e' = (u, v)$ belongs to positive example edges' type's set; in case it does belong, the edge becomes a negative example edge. Statistics regarding all positive and negative examples for CT an AC are shown in Tables 1 and 2, respectively.

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 at random. Hence, positive and negative sets will also have the same number of examples. However, for a type of edge with fewer examples, this might imply in undersampling for this type of edge. The resulting datasets for both games are shown in Table 3.

5.3. 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, the 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 embeddings with all the available architectures.

Each architecture performance and their resulting embeddings are evaluated and discussed in Section 6 through several experiments and simulations.

5.4. Classifier training

Classifier training is realized using positive and negative example sets defined in the preprocessing step and node embeddings defined on the previous step. We adapted 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 the 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 [40], a well-known Machine Learning library in python that 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.

6. Experimental results

In this section, we describe how our solution may answer the proposed research questions through the experiments described in Section 5.

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

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

Table 4
Averaged time (seconds) duration of CT's embedding generation step.

LSTM	MaxPool	MeanPool	Mean	GCN
61.236	25.823	24.876	22.367	20.175

Table 5
Averaged time (seconds) duration of CT's classifier training.

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 6
Averaged results of CT classifiers in a 7-fold cross-validation setting.

Approach	Precision	Recall	F1
Without Provenance + SVM	0.249	0.293	0.261
Without Provenance + MLP	0.381	0.382	0.381
Without Provenances + SGD	0.545	0.542	0.534
PingUMiL.LSTM + Mult + MLP(100,1)	0.673	0.663	0.664
PingUMiL.LSTM + Mult + MLP(100,2)	0.641	0.654	0.638
PingUMiL.LSTM + Cat + MLP(100,1)	0.664	0.662	0.661
PingUMiL.LSTM + Cat + MLP(512,1)	0.672	0.656	0.662
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
PingUMiL.GCN + Mult + MLP(100,2)	0.549	0.567	0.538

hidden layer with 100 units.

In our experiments, we measure precision, recall, and f1-score [43] for encoded edges of both case studies in a stratified k-fold cross-validation setting [44]. 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 case studies, CT is divided into 7 folds while AC is divided into 4 folds. We opted for these respective number of folds so that each fold from both game prototypes have approximately the same amount of edges.

We compare the generated classifiers from PingUMiL against traditional classification methods using features without provenance and the same classifiers as before (SVM, MLP, and SGD). We call features without provenance the attribute values attached to each node in the original provenance graph, such as speed, acceleration, position, but without the provenance graph structure and, therefore, without the relational nature of the data. In this scenario, each node is an entry in the game log. The features without provenance were fed to these methods in order to provide a baseline. The primary motivation of this baseline is to investigate whether the use of embeddings enhance edge detection over features without provenance and answer the research questions Q2. We also use these experiments to answer research question Q1 since both classical and graph representation approaches are applied for influence detection.

6.1. Car Tutorial

As aforementioned, Car Tutorial example edges were split into 7 folds and fed into several different learning settings, leading to more than **60** models. Tables 4 and 5 presents averaged time duration of embedding generation (for all CT nodes) and some classifiers' training step, respectively, which shows that GraphSAGE is capable of processing a vast number of nodes, and learning embeddings for them in, at most, 62 s. Each combination of fold and model is trained 50 times, and their performance is measured and averaged for each fold and each model. Table 6 presents some results for the generated models' analysis. All averaged metrics presented in Table 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.

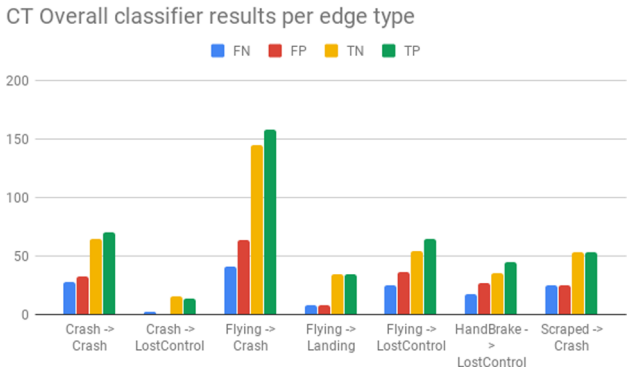


Fig. 13. Overall prediction results per edge type in CT graphs.

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. A similar 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. The 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.

In Fig. 14, we compare results the performance of three models per fold. First, PingUMiL.LSTM + Cat + MLP(100,2), which presented the best overall metrics in Table 6. Then, PingUMiL.LSTM + Cat + MLP (100,1), which achieved best average recall and F1 with 72,2% and 71,9%, respectively, also in fold 1. Finally, PingUMiL.LSTM + Mult + MLP(100,1), which achieved the best average precision in fold 1 with 72,3%. It' is possible to perceive that the distribution of the metrics among folds of the Cat models are very similar. The MLP(100,1) architecture's values seem to vary less, which leads to the intuition that adding more layers to the MLP does not improve performance significantly. Comparing Cat models and the Mult model, it is possible to notice that the latter tends to present less variance, except for fold 7. 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 model training. Therefore, we intend to investigate it further soon.

After evaluating the overall performance of the generated models, an in-depth investigation regarding the quality of predictions per type of edges was realized. For this investigation, the output of a generated PingUMiL.LSTM + Cat + MLP(100,1) model for each test fold is taken randomly. We took the number of false negatives (FN), which are valid indirect edges predicted as invalid indirect edges; false positives (FP), i.e., invalid indirect edges predicted as valid indirect edges; true negatives (TN), invalid indirect edges predicted as invalid indirect edges; and true positives (TP), valid indirect edges predicted as valid indirect edges.

It is possible to observe in the Fig. 13, in which the horizontal axis lists edge types and the vertical axis represents the number of edges predicted as FN, FP, TN, and TP, that a trend holds for most edge types: most edges are correctly classified (TP and TN outnumbers FP and FP)

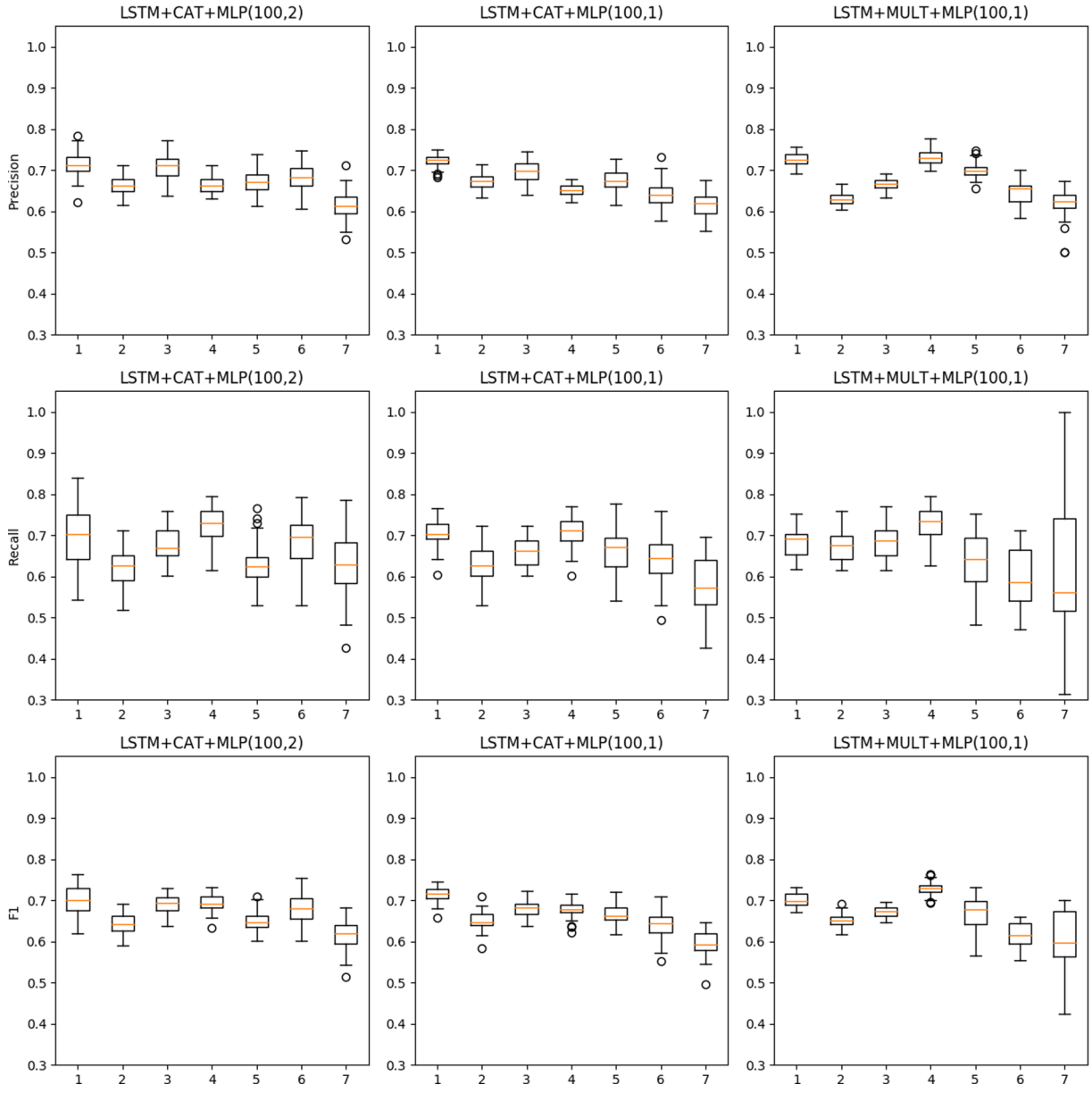


Fig. 14. Boxplots of the Precision, Recall and F1 metrics for 3 CT PingUMiL models per fold.

Table 7
Averaged time (seconds) duration of AC's embedding generation step.

LSTM	MaxPool	MeanPool	Mean	GCN
58.839	24.061	20.936	18.328	17.091

Table 8
Averaged time (seconds) duration of AC's classifier training.

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

Table 9
Averaged results of AC classifiers in a 4-fold cross-validation setting.

Approach	Precision	Recall	F1
Without Provenance + SVM	0.249	0.334	0.273
Without Provenance + MLP	0.394	0.394	0.394
Without Provenance + SGD	0.596	0.578	0.575
PingUMiL.LSTM + Cat + MLP(100,1)	0.696	0.7	0.697
PingUMiL.LSTM + Cat + MLP(100,2)	0.695	0.69	0.692
PingUMiL.LSTM + Cat + MLP(256,1)	**0.696**	**0.702**	**0.698**
PingUMiL.LSTM + Cat + SGD	0.662	0.613	0.608
PingUMiL.MeanPool + Mult + MLP(100,1)	0.646	0.638	0.641
PingUMiL.Mean + Cat + MLP(100,1)	0.665	0.628	0.646
PingUMiL.GCN + Cat + MLP(256,1)	0.695	0.663	0.678

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 valid influence edge incorrectly predicted and, therefore, not inserted in the graph enhanced by PingUMiL. On the other hand, a false positive is an edge that (1) makes no sense in its neighborhood context and can be ignored by the game analyst or (2) represents an unforeseen influence with regard to edge examples fed to

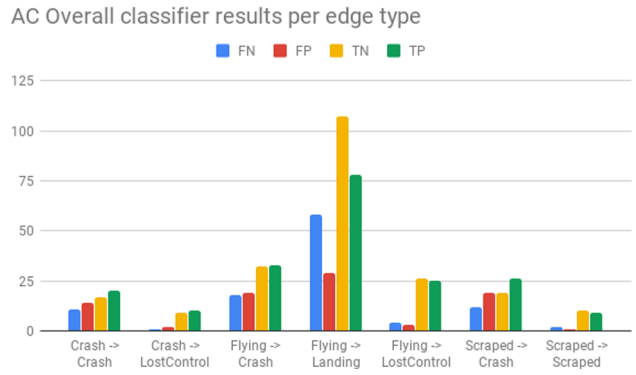


Fig. 15. Overall classification results per edge type in AC graphs.

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.

6.2. Arcade Car

The Arcade Car example edges were split into 4 folds and fed into the same learning settings used in CT experiments. Tables 7 and 8 presents averaged time duration of embedding generation (for all AC nodes) and some classifiers' training step, respectively. Similarly, each combination of fold and model is trained 50 times, and their performance is measured and averaged for each fold and each model. Table 9 presents some results for the generated models' analysis. All averaged metrics presented in Table 9 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.

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

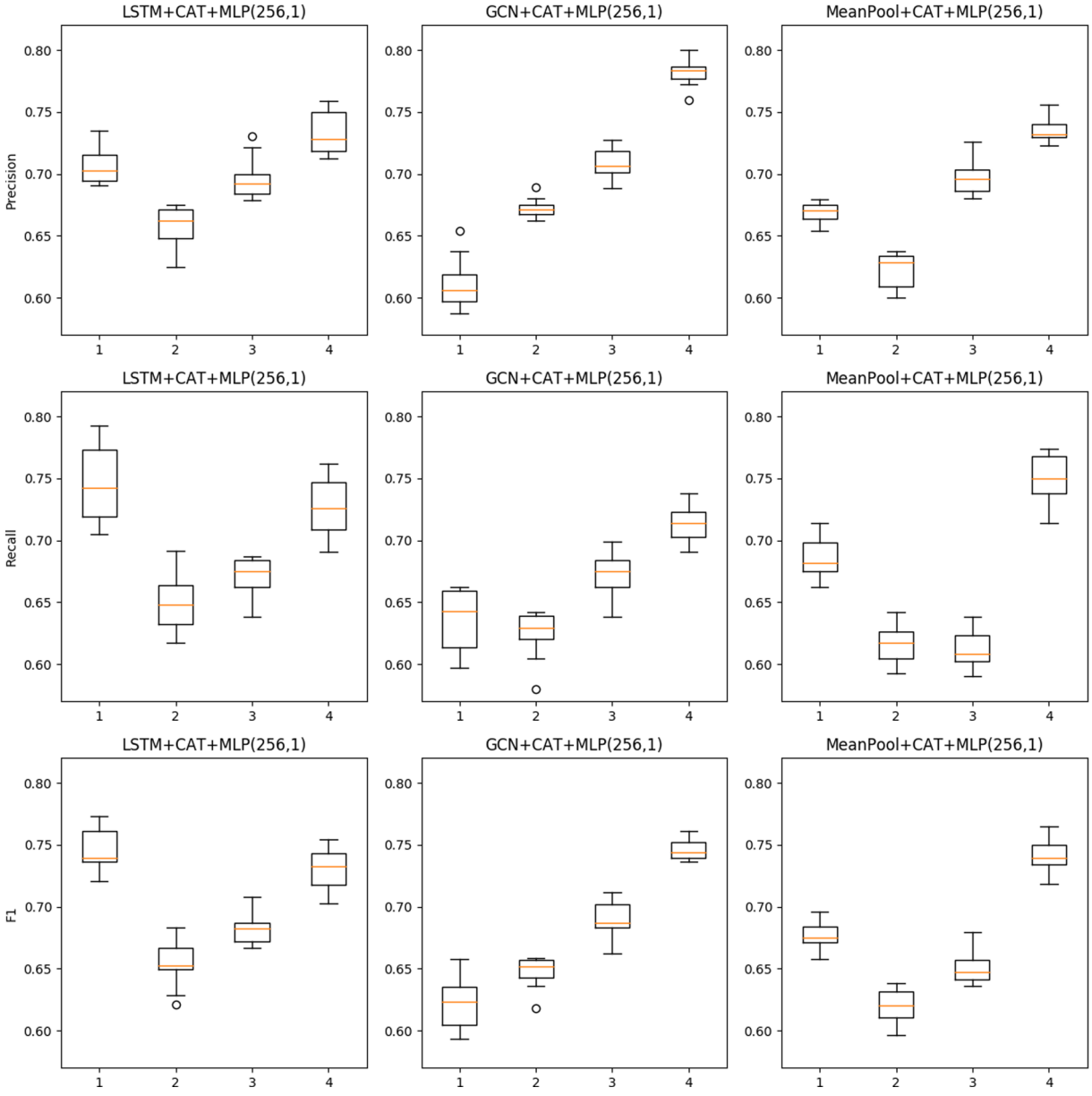


Fig. 16. Boxplots of the Precision, Recall and F1 metrics for 3 AC PingUMiL models per fold.

baseline methods.

Notice that most of the results in Table 9 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 PingUMiL.MeanPool + Mult + MLP(100,1) achieved the highest performance using a model with the "Mult" edge encoding function.

The performance reached by the 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. In Fig. 16, we compare results from all metrics for models PingUMiL.LSTM + Mult + MLP(256,1), which achieved best averaged result in Table 9, PingUMiL.GCN + Cat + MLP(256,1), which achieved the best average precision and F1 in fold 4 with 77,1% and 74,3%, respectively, on fold 4, and PingUMiL.MeanPool + Cat + MLP(256,1), whose average recall achieved 74% in fold 4. In general, the boxplots corroborates with the balanced performance achieved by LSTM. For both Precision and Recall, even though GCN and MeanPool models, respectively, achieved the best average value in fold 4, the distribution per fold shows that LSTM is still more reliable. Nevertheless, 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, especially for LSTM models. This repeated behavior confirms that fold composition influences the model's performance and must be investigated in the near future.

Observing the results presented in both Tables 6 and 9, it is possible to answer the research questions Q1 and Q2. For Q1, Machine Learning techniques can detect influence between game components represented as edges in a game provenance graph, given that the models achieved above 77% average precision. For Q2, PingUMiL best performance presents a gain of at least 10% over classical machine learning approaches. These performance results points that game data structured in provenance graphs are capable of providing better information to the model since every embedded node aggregates itself and its neighborhood attributes during the model training step.

Similar to the previous subsection, we investigate the model's performance regarding the quality of predictions per type of edges using a generated PingUMiL.GCN + Cat + MLP(256,1) model for each test fold. Fig. 15 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 the results in CT, AC presents the smallest gap between incorrectly detected examples (FN and FP) and correctly detected examples (TN and TP). In Flying → Landing, Flying → LostControl, and Scraped → Scraped edges, the number of TN is higher than TP, i.e., the models managed to correctly predict invalid examples easier than valid ones. Consequently, the number of FN is higher than FP, which is dangerous for the reasons already discussed in the previous subsection: a false negative edge is a valid influence edge incorrectly predicted and, therefore, not inserted in the graph enhanced by PingUMiL, which leads to missing information to Game Analytics tasks.

7. Conclusion

This work, to the best of our knowledge, is the first attempt in the literature to combine machine 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 for edge detection 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. 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 without provenance with a gain of at least 10% on precision, recall, and F1 scores. Since we realized experiments in similar games, we intend to assess the generalization capabilities of such models in the near future.

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 soon. Even though GraphSAGE is a powerful, usable, and expressive embedding generation tool, it still lacks heterogenous nodes support. It is relevant to mention that the PingUMiL framework should benefit from upcoming improvements and advances in graph representation learning techniques.

One of the main assumptions of this research is the cost and effort of manual addition of edges and implementation of methods for capturing indirect influences being higher than using our edge annotation based machine learning approach. As future work, we intend to scientifically confirm this assumption, by comparing the performance of a model and human experts in terms of time and soundness.

Also, experiments conducted in our case study game prototypes intended to assess the framework and evaluate its potential capabilities. The case study games, in their turn, do not reflect the complexity of the guiding example presented in the paper or real-world game development industry. Instantiating the PingUMiL framework in this type of scenario is necessary to obtain stronger evidence of this paper's contribution.

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.

Declaration of Competing Interest

The authors declared that there is no conflict of interest.

Acknowledgment

The authors would like to thank CAPES, CNPq, and FAPERJ for financial support.

References

[1] G. Andrade, G. Ramalho, H. Santana, V. Corruble, Extending reinforcement learning to provide dynamic game balancing, Proceedings of the Workshop on Reasoning, Representation, and Learning in Computer Games, 19th International Joint Conference on Artificial Intelligence (IJCAI), 2005, pp. 7–12.
[2] V. Volz, G. Rudolph, B. Naujoks, Demonstrating the feasibility of automatic game balancing, Proceedings of the Genetic and Evolutionary Computation Conference 2016, ACM, 2016, pp. 269–276.
[3] T. Mahlmann, A. Drachen, J. Togelius, A. Canossa, G.N. Yannakakis, Predicting player behavior in tomb raider: Underworld, Computational Intelligence and Games (CIG), 2010 IEEE Symposium on, IEEE, 2010, pp. 178–185.
[4] A. Tychsen, A. Canossa, Defining personas in games using metrics, Proceedings of the 2008 Conference on Future Play: Research, Play, Share, ACM, 2008, pp. 73–80.
[5] A. Zook, B. Harrison, M.O. Riedl, Monte-carlo tree search for simulation-based strategy analysis, Proceedings of the 10th Conference on the Foundations of Digital Games, 2015.
[6] P. Guardini, P. Maninetti, Better game experience through game metrics: a rally videogame case study, Game Analytics, Springer, 2013, pp. 325–361.
[7] T. Fields, B. Cotton, Social Game Design: Monetization Methods and Mechanics, CRC Press, 2011.
[8] M. Viljanen, A. Airola, A.-M. Majanoja, J. Heikkonen, T. Pahikkala, Measuring player retention and monetization using the mean cumulative function, 2017. arXiv preprint arXiv: 1709.06737.

[9] M.S. El-Nasr, A. Drachen, A. Canossa, Game Analytics, Springer, 2016.
[10] C. Bauckhage, K. Kersting, R. Sifa, C. Thurau, A. Drachen, A. Canossa, How players lose interest in playing a game: an empirical study based on distributions of total playing times, Computational Intelligence and Games (CIG), 2012 IEEE Conference on, IEEE, 2012, pp. 139–146.
[11] T. Kohwalter, E. Clua, L. Murta, Provenance in games, in: Brazilian Symposium on Games and Digital Entertainment (SBGAMES), 2012, p. 11.
[12] T.C. Kohwalter, E.G. Clua, L.G. Murta, Game flux analysis with provenance, Advances in Computer Entertainment, Springer, 2013, pp. 320–331.
[13] T.C. Kohwalter, L.G.P. Murta, E.W.G. Clua, Capturing game telemetry with provenance, 2017 16th Brazilian Symposium on Computer Games and Digital Entertainment (SBGames), IEEE, 2017, pp. 66–75.
[14] T.C. Kohwalter, F.M. de Azeredo Figueira, E.A. de Lima Serdeiro, J.R. da Silva Junior, L.G.P. Murta, E.W.G. Clua, Understanding game sessions through provenance, Entertain. Comput. 27 (2018) 110–127.
[15] W.L. Hamilton, R. Ying, J. Leskovec, Representation learning on graphs: Methods and applications, 2017. arXiv preprint arXiv: 1709.05584.
[16] E. Alpaydin, Introduction to Machine Learning, MIT Press, 2009.
[17] M. Schubert, A. Drachen, T. Mahlmann, Esports analytics through encounter detection other sports, 2016.
[18] F. Block, V. Hodge, S. Hobson, N. Sephton, S. Devlin, M.F. Ursu, A. Drachen, P.I. Cowling, Narrative bytes: data-driven content production in esports, Proceedings of the 2018 ACM International Conference on Interactive Experiences for TV and Online Video, ACM, 2018, pp. 29–41.
[19] M. Freire, Á. Serrano-Laguna, B.M. Iglesias, I. Martínez-Ortiz, P. Moreno-Ger, B. Fernández-Manjón, Game learning analytics: learning analytics for serious games, Learning, Design, and Technology, Springer, 2016, pp. 1–29.
[20] M.D. Kickmeier-Rust, Predicting learning performance in serious games, Joint International Conference on Serious Games, Springer, 2018, pp. 133–144.
[21] L. Moreau, B. Clifford, J. Freire, J. Futrelle, Y. Gil, P. Groth, N. Kwasnikowska, S. Miles, P. Missier, J. Myers, et al., The open provenance model core specification (v1. 1), Future Gener. Comput. Syst. 27 (2011) 743–756.
[22] T.C. Kohwalter, E.W.G. Clua, L.G.P. Murta, Reinforcing software engineering learning through provenance, 2014 Brazilian Symposium on Software Engineering (SBES), IEEE, 2014, pp. 131–140.
[23] T. Kohwalter, T. Oliveira, J. Freire, E. Clua, L. Murta, Prov viewer: a graph-based visualization tool for interactive exploration of provenance data, International Provenance and Annotation Workshop, Springer, 2016, pp. 71–82.
[24] F. Scarselli, M. Gori, A.C. Tsoi, M. Hagenbuchner, G. Monfardini, The graph neural network model, IEEE Trans. Neural Netw. 20 (2009) 61–80.
[25] B. Perozzi, R. Al-Rfou, S. Skiena, Deepwalk: Online learning of social representations, Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, ACM, 2014, pp. 701–710.
[26] M. Ou, P. Cui, J. Pei, Z. Zhang, W. Zhu, Asymmetric transitivity preserving graph embedding, Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, ACM, 2016, pp. 1105–1114.
[27] A. Grover, J. Leskovec, node2vec: scalable feature learning for networks, in, Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, ACM, 2016, pp. 855–864.
[28] B.P. Chamberlain, J. Clough, M.P. Deisenroth, Neural embeddings of graphs in hyperbolic space, 2017, arXiv preprint arXiv: 1705.10359.
[29] W. Hamilton, Z. Ying, J. Leskovec, Inductive representation learning on large graphs, Advances in Neural Information Processing Systems, 2017, pp. 1025–1035.
[30] Y. Li, D. Tarlow, M. Brockschmidt, R. Zemel, Gated graph sequence neural networks, International Conference on Learning Representations (ICLR), 2016.
[31] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Liò, Y. Bengio, Graph attention networks, 2017. arXiv preprint arXiv: 1710.10903.
[32] D. Wang, P. Cui, W. Zhu, Structural deep network embedding, Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, ACM, 2016, pp. 1225–1234.
[33] T.N. Kipf, M. Welling, Semi-supervised classification with graph convolutional networks, 2016. arXiv preprint arXiv: 1609.02907.
[34] M. Belkin, P. Niyogi, Laplacian eigenmaps and spectral techniques for embedding and clustering, Advances in Neural Information Processing Systems, 2002, pp. 585–591.
[35] T. Pham, T. Tran, D.Q. Phung, S. Venkatesh, Column networks for collective classification, in: AAAI, 2017, pp. 2485–2491.
[36] Y. Dong, N.V. Chawla, A. Swami, metapath2vec: scalable representation learning for heterogeneous networks, Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, ACM, 2017, pp. 135–144.
[37] M. Zitnik, J. Leskovec, Predicting multicellular function through multi-layer tissue networks, Bioinformatics 33 (2017) i190–i198.
[38] L.F. Ribeiro, P.H. Saverese, D.R. Figueiredo, struc2vec: learning node representations from structural identity, Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, ACM, 2017, pp. 385–394.
[39] X. Meng, J. Bradley, B. Yavuz, E. Sparks, S. Venkataraman, D. Liu, J. Freeman, D. Tsai, M. Amde, S. Owen, et al., Mllib: machine learning in apache spark, J. Mach. Learn. Res. 17 (2016) 1235–1241.
[40] D. Cournapeau, Sci-kit learn, Machine Learning in Python. online, 2015. URL http://scikit-learn.org. (cit. 8.5. 2017).
[41] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, I.H. Witten, The weka data mining software: an update, ACM SIGKDD Explor. Newsletter 11 (2009) 10–18.
[42] S. Makeev, Arcade car physics – vehicle simulation for unity3d, 2018, https://github.com/SergeyMakeev/ArcadeCarPhysics.
[43] D.L. Olson, D. Delen, Advanced Data Mining Techniques, Springer Science & Business Media, 2008.
[44] R. Kohavi et al., A study of cross-validation and bootstrap for accuracy estimation and model selection, in: Ijcai, vol. 14, Montreal, Canada, 1995, pp. 1137–1145.