Representing Player Behaviour via Graph Embedding Techniques: A Case Study in Dota 2

1st Jinay Shah
RISE Research Group
School of Information Technology
Carleton University
Ottawa, Canada
jinay.shah@carleton.ca

2nd David Thue
RISE Research Group
School of Information Technology
Carleton University
Ottawa, Canada
david.thue@carleton.ca

***Abstract*—We explore the use of graph embedding techniques to represent the player behaviour that is expressed in the logs of video games. While such logs hold data that could be useful for personalization, the data is often poorly structured for use with Artificial Intelligence systems and its dimensionality is often high. By using a graph to structure the logs and applying embedding techniques to reduce their dimensionality, a compact vector representation can be obtained that preserves some of their semantics. To explore the potential value of this approach, we obtained gameplay logs from over 3000 matches of Defense of the Ancients 2 (Dota 2) and compared 13 parameter variations of three different embedding techniques: NODE2VEC, LINE, and TGN. Our analysis considers the effects of embedded vector size, dataset size, a step size used for updating vectors as a game proceeds, and different types of player interaction. The results show that NODE2VEC outperforms the other techniques on 7 of the 13 variations that we tested, and that removing one type of player interaction can make it easier to predict the others.**

***Index Terms*—Player Modelling, Graph Embedding, Dota 2**

I. Introduction

Machine Learning (ML) models generally need their input data to be given in numerical form. Because of this, it is essential to convert real-world data like text, images, videos, and graphs into a numerical format (i.e., a vector) before supplying them to an ML model. Minimizing the size of such vectors (and keeping their representation of the data dense) helps to keep the ML model small and avoid overfitting.

What if we could embed player behaviour in a dense vector space? For one, knowing that a vector represented a particular behaviour would make it easier to recognize that behaviour. Behaviour recognition is a key challenge of personalization, which can have a significant impact on player engagement [1]. With reliable and automated behaviour recognition, a player's play style could be identified and used to tailor a game's progression [2], [3]. Furthermore, with sufficient data, it might be possible to pre-train models that embed more generic behaviours in a game-independent way, and then fine-tune on a per-game basis to embed more game-specific behaviours. Such a scheme might facilitate the transfer of player behaviour data across different games, while also reducing the computational cost of training models for multiple games.

The authors acknowledge the support of the Natural Sciences and Engineering Research Council of Canada (NSERC), Grant #2020-06502.

Previous methods for modelling player behaviour using ML have relied heavily on handcrafted features that are highly game-specific [4], [5]. Furthermore, Eggert et al.'s classification of *Defense of the Ancients 2* players also required extensive manual labelling of gameplay logs, which limits the general viability of their approach [4], [6]. Recent applications of ML to modelling for games have sought to encode in-game objects [7] or predict player emotions [8], but they did not attempt to represent player behaviour.

Embedding techniques are methods for encoding objects such as words, images, or graphs as vectors in a dense, low-dimensional space [9]–[12], and *graph embedding techniques* are those that particularly accept data represented as a graph (i.e., nodes connected by edges) as their inputs [13]–[15]. In this paper, we propose a way to apply and evaluate graph embedding techniques as methods for representing player behaviour. Compared to previous modelling methods, our approach is game-agnostic, requires no manual labelling of logs, and avoids a need for handcrafted features. To demonstrate our approach, we situate our work in the context of a popular commercial video game (*Defense of the Ancients 2*) and attempt to represent the behaviour of players therein. Using our general method for converting gameplay logs into a graph of player interactions, we apply three graph embedding techniques and assess their value: NODE2VEC, LINE, and TGN [13]–[15]. To establish a metric of success, we adapt a link prediction task from the graph embedding literature [16] to the domain of gameplay logs, and assess how well the embedded vectors from each technique can inform predictions of future player interactions. We evaluate each technique across a variety of parameterizations, to help characterize their behaviour and inform future work. Beyond this characterization, our experiments reveal that the relative frequency of different types of interaction might influence how fully any lower-frequency interactions get represented in the embedded vectors.

Before proceeding with the rest of the paper, we briefly introduce embedding techniques in general and then review the three graph embedding techniques that we tested. We then give an overview of *Defense of the Ancients 2* (*Dota 2*).

A. *Embedding Techniques*

The goal of an embedding technique is to encode the semantic meaning of some given objects (e.g., text, images, graphs), which can simplify comparison via ML algorithms. For example, word2vec embeds words into a low-dimensional space using a neural network [9]. The addition and subtraction of these vectors result in new representations in the vector space (e.g., adding the vectors for a car and the colour red can construct a vector representation for a red car). Mordatch showed that simple concepts can be successfully embedded as vectors [10], [11], and Du, Li, and Mordatch extended the same idea to represent images as vectors [12].

B. *Graph Embedding Techniques*

Graph embedding techniques take a graph as input, where a *graph* is defined as a set of nodes and a set of edges, where each edge connects two nodes as neighbours in the graph. For the techniques that we explored for this work, each node of the given graph is embedded as a separate vector in a shared vector space. Specifically, we investigated three techniques: NODE2VEC [13], LINE (Large-scale Information Network Embedding) [14], and TGN (Temporal Graph Network) [15]. We chose these techniques due to their promising results in other embedding tasks and their easily accessible and executable source code. To help keep the paper self-contained, we briefly review the operation of each technique.

1) NODE2VEC*:* The word embedding method WORD2VEC successfully embeds words in a vector space based on how they co-occur in the sentences of a corpus of training text [9]. The words can be of any language and use unique identifiers, since the only information relevant for WORD2VEC is which words appear together in a sentence. NODE2VEC [13] leverages this flexibility of WORD2VEC by generating "sentences" as strings of nodes that are visited by random walks across the graph. Nodes that are connected by edges in the graph appear adjacent to one another in these generated sentences. A corpus of generated sentences is then used to train a neural network (similarly to WORD2VEC) and provide vector representations of the nodes's connections in the graph.

2) LINE*:* The core premise of LINE (Large-scale Information Network Embedding) is that two nodes that are connected in the graph should be closer to one another in the vector space than nodes that are not connected [14]. LINE uses a fully connected, deep neural network to encode each graph node and its connections to other nodes as a vector. The encoded vectors of two nodes are compared, and the weights of the network are tuned to minimize/maximize the distance between nodes that are/aren't connected.

3) TGN*:* TGN (Temporal Graph Network) works by building node vectors one interaction at a time [15], and thus supports building vectors dynamically as the interactions happen. In the text domain, an interaction might be a co-occurrence of two words in the same sentence, where each word is a node in the graph. In a game, it might be a co-occurrence of two game entities in the log of a single in-game event, where each entity is a node in the graph. At a high level, TGN maintains a "memory" vector for each graph node that it detects in a stream of interactions; this vector serves as a compact representation of the node's history of interactions. To produce an embedded vector for a given node n, TGN combines n's memory vector with the memory vectors of other nodes that n has recently interacted with. TGN's functions for updating the memory and for embedding nodes as vectors are implemented as neural networks, which are trained using a loss function based on predicting edges in the graph (future node interactions).

C. *Defense of the Ancients 2*

A full description of the game that we used for testing is beyond the scope of this paper, but a brief introduction should be sufficient to understand the context of our work. *Defense of the Ancients 2* [6] is a competitive, multiplayer computer game in which two teams of players compete for control of a small geographical area. Its title is commonly shortened to "*Dota 2*". Typical gameplay consists of a match between two teams of five heroes, where each player chooses their desired hero before the match begins and controls that hero during the match. A *hero* is a player-controlled, in-game character that can move, attack, use a variety of special abilities, collect and use *runes* (power-ups), and buy and use special items. Heroes differ in terms of gameplay statistics (e.g., health, damage), special abilities, and artwork, and at the time of writing, *Dota 2* offered more than 100 unique heroes to choose from. Each team begins in a base on opposite sides of the area and works to progress across the area and destroy the opposing team's base. Each player can earn in-game gold via combat, and they can spend this gold to buy items for later use.

A limitation of our strategy for assessing embedding techniques is that it requires gameplay logs that have timestamps: one for each entry that describes a player interacting with an entity in the game. As sources for gameplay logs, we considered four games that had large player bases and offered open player data: *Dota 2* [6], *PlayerUnknown's Battlegrounds (PUBG)* [17], *StarCraft II* [18] and *League of Legends* [19]. At the time of our analysis, only *Dota 2* and *PUBG* offered data that met our timestamp requirement. Of those two, we selected *Dota 2* due to the convenient access to data afforded by the OpenDota API [20]. In the available dataset from *Dota 2*, logs from 3281 play sessions contained the timestamps that we needed for our study. We used all 3281 logs.

II. Problem Formulation

The overarching goal of our work is to better understand how different graph embedding techniques perform on player behaviour data. For this paper, we sought to characterize the performance of three techniques (NODE2VEC [13], LINE [14], and TGN [15]) when representing data from the logs of a computer game. To focus our work, we pursued two desirable properties for such techniques.

First, they must produce vectors that each represent their associated player behaviour in a meaningful way. Following Xu et al. [16], we quantify this notion by training three linear classifiers (each using vectors produced by only one of the three techniques) and measuring the accuracy of each classifier. Each classifier must repeatedly predict whether a given pair of game entities will interact before the game ends. We posit that higher prediction accuracies for this task provide stronger evidence that a method's vectors represent meaningful information. Intuitively, the more accurately that some vectors can be used to predict a game's future, the more likely it is that those vectors represent meaningful game information.

Second, a graph embedding technique must produce vectors that require less data than the raw behaviour logs that they were produced from, to minimize storage costs and vector computation time. We compare the storage footprints of both our logs from *Dota 2* and the vectors that our three tested techniques produced.

III. Related Work

Khameneh and Guzdial used neural networks to derive embeddings of game entities based on their physical attributes, which included their size, velocity, and location [7]. They aimed to capture information about game dynamics in the resulting latent space, toward supporting procedural content generation tasks that use machine learning. Starting from roughly 100 frames of gameplay video from two Atari games, they first extracted a ruleset for each game using Guzdial, Li, and Riedl's Game Engine Search algorithm [21]. From these rulesets, they identified possible attribute configurations of each game's entities. Given these possible configurations, they represented each entity using a one-hot vector of 1600 values, and then trained a variational auto-encoder to embed each 1600-D input vector into a 25-D vector space.

Our work differs from Khameneh and Guzdial's in two key ways. First, the starting point of our data is text logs of gameplay, rather than video of gameplay. While the availability of gameplay video as a data source is higher, it is nevertheless useful to develop methods that can leverage the structured information contained within text logs, given that such logs are often available. Second, the data that we aim to embed represents player behaviour – not the physical attributes of game entities. While the Game Engine Search algorithm does seem capable of accounting for player actions as part of its rulesets [21], Khameneh and Guzdial did not describe using any rules that would have done so. We thus assume that their vectors did not represent player behaviour.

Eggert et al. tested the performance of different methods of classifying the behaviour of *Dota 2* players, each into one of nine custom-defined roles [4]. They identified 18 data features to use for the classification task, eight of which were available directly from gameplay logs. The ten remaining features required manual annotation to obtain, and they used crowd-sourcing in which participants watched and labeled a replay of every game in their dataset. Manual annotation is time consuming and potentially prone to errors, and so instead our analysis relies solely on data that can be parsed automatically from *Dota 2* replay files. Furthermore, our criteria for selecting data features was more general: if a feature described an interaction between a player and some other entity in the game, we included that feature in our selection.

Pirker et al. used Sankey diagrams to model how players of *Just Cause 2* [22] transitioned between different archetypes while playing [5]. They derived their clusters from a set of features that they hand-picked from their dataset. Our method avoids any need to hand-pick features.

Makantasis, Liapis, and Yannakakis combined several neural networks to predict a player's arousal based on the video and audio signal from a game that the player plays [8]. While having a model of player arousal could be useful in predicting a player's behaviour, such a model does not aim to *represent* that behaviour, which is what we do in the present work.

Rabii and Cook applied WORD2VEC [9] (upon which NODE2VEC is based) to embed data from a large collection of logs of players playing chess [23]. Rather than representing how players might behave, they sought to represent the dynamics of Chess in the resulting embeddings, and they did not compare WORD2VEC to any other techniques.

We are not aware of any prior work that has attempted to deploy and characterize graph embedding techniques in the context of representing player behaviour.

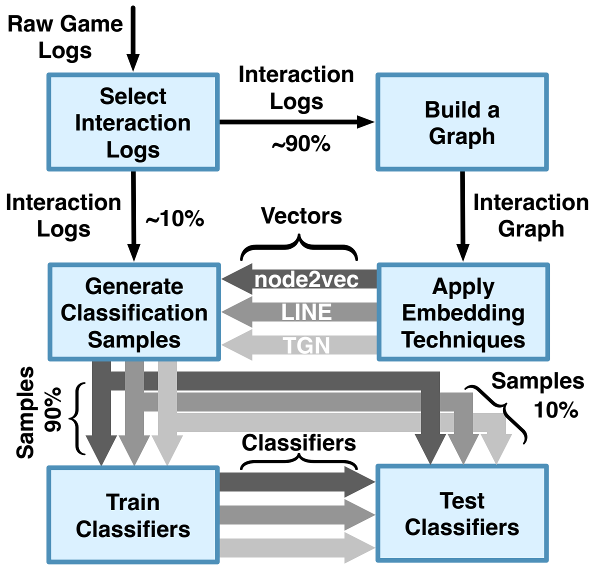


Fig. 1. Our framework for experimentation. Boxes represent computation while arrows represent data. The shades of thick arrows distinguish between data that was related to the vectors produced by NODE2VEC (darkest shade), LINE, and TGN (lightest shade).

IV. Proposed Approach

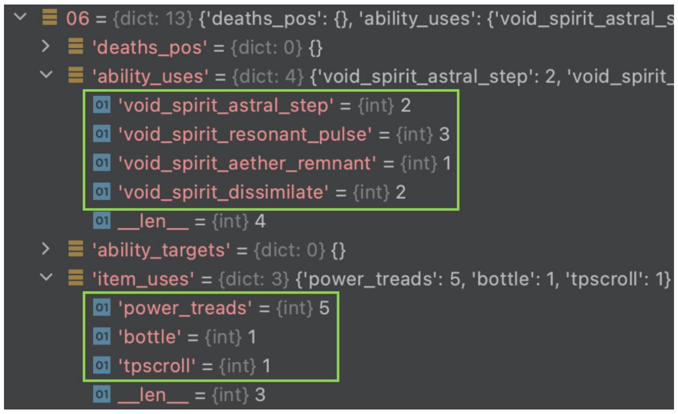
To characterize how well graph embedding techniques can represent player behaviour from gameplay logs, we developed the framework for experimentation that appears in Figure 1. Starting from raw logs that span a large number of play sessions, we first select all log entries that represent player interactions (with the game environment or with other players). Next, we use the selected logs from ∼90% of the sessions to build a graph of player interactions (see Section IV-A), withholding a randomly drawn ∼10% for generating unseen test cases. We use the graph as the input to each of our tested embedding techniques (Section I-B), where each technique produces embedded vectors: one for every node in the graph. We evaluate each technique's vectors by training and testing a simple classifier (one per technique) on a prediction task: given a history of all players' interactions up to some random time point within a play session, predict whether two randomly chosen entities will interact before the play session ends. We generate the samples to train and test these classifiers using the ∼10% of play sessions that we withheld from building the graph, as doing so helps us assess whether the information that the vectors represent can be generalized to novel play sessions (see Section IV-B). We train and test each classifier using the generated samples via 10-fold cross-validation. We explain our choice of classifier in Section IV-C.

A. Building a Graph from Interactions

Gameplay logs are often stored as text. While word embedding techniques for text have seen substantial popularity and success [9], [24], they were designed to work on data that is less structured (e.g., natural sentences) than what is contained in gameplay logs. Li et al. found that graph embedding techniques offer a promising way to take advantage of such structured information [25], modelling data features as nodes in a graph where edges represent interactions between those features. We adopt this strategy in our work; a feature identifies an in-game entity as a label in a log entry, and each log entry that associates two entities at a particular point in time constitutes an interaction between those features/entities. For simplicity, we only model binary interactions.

Since we aim to capture player behaviour, we build a graph using only log entries that have at least one entity that is controlled by a player (e.g., a hero in *Dota 2*). Having selected the log entries that meet this criterion, we build the graph as follows. First, create a node in the graph for each entity that is identified in at least one of the selected log entries. Using our logs from *Dota 2*, the result is a graph that contains a node for each of the game's playable heroes, each of the items that a hero can purchase and use, and each of the abilities and runes that a hero can use, amounting to roughly 1000 nodes.

Next, we must consider whether we are building a graph for an embedding technique that works on static graphs (like NODE2VEC or LINE), or dynamic graphs (like TGN). If building a static graph, then for each pair of entities (m, n) that appear together in at least one of the selected log entries, we add an edge to the graph between m's and n's nodes (provided that the edge does not already exist in the graph). Note that static graphs fail to capture any information about repeated interactions between two entities, because only the first interaction has any effect on the constructed graph. If building a dynamic graph, then for every selected log entry in which a pair of entities (m, n) appear together at some particular timestamp, we add an edge to the graph between m's and n's nodes that is labelled with that timestamp. In such a dynamic (multi-)graph, every interaction between two entities gets represented in the graph, because two nodes can have multiple edges between them. Using our *Dota 2* logs, the graph for NODE2VEC and LINE gains roughly 32k edges, while the graph for TGN gains roughly 124k edges.



```
06 = {dict: 13} {'deaths_pos': {}, 'ability_uses': {'void_spirit_astral_s
  'deaths_pos' = {dict: 0} {}
  'ability_uses' = {dict: 4} {'void_spirit_astral_step': 2, 'void_spirit_
    'void_spirit_astral_step' = {int} 2
    'void_spirit_resonant_pulse' = {int} 3
    'void_spirit_aether_remnant' = {int} 1
    'void_spirit_dissimilate' = {int} 2
    __len__ = {int} 4
  'ability_targets' = {dict: 0} {}
  'item_uses' = {dict: 3} {'power_treads': 5, 'bottle': 1, 'tpscroll': 1}
    'power_treads' = {int} 5
    'bottle' = {int} 1
    'tpscroll' = {int} 1
    __len__ = {int} 3
```

Fig. 2. Part of a *Dota 2* log sourced from the OpenDota API [20], with some of Player 6's ability uses and item uses highlighted by green rectangles.

To help illustrate the graph building process, consider the partial log data shown in Figure 2. This data would result in eight nodes being added to the graph, assuming that they had not already been added while processing an earlier part of the log. These nodes would include one for player 6 (`06` in the figure), one for each of the four `void_spirit` abilities shown, and one for each of the three items shown. For the graph used by NODE2VEC and LINE, seven edges would be added, if they were not already in the graph: one for each connection between player 6 and each of the other seven nodes. For the graph used by TGN, the seven edges would be added (with timestamps) regardless of whether or not any of them were already in the graph with earlier timestamps.

B. Generating Samples for Classification

After applying any graph embedding technique to a graph built from player logs, we obtain a set of embedded vectors – one for each node in the graph. At a high level, we test the quality of these vectors by using them to train and test a classifier; the classifier tries to predict whether any given pair of game entities will interact before a given play session ends, and after the point in time that the vectors describe.

To describe a point in time during a play session using our embedded vectors, we update the vectors to account for all of the entity interactions that can be observed from the log of the given play session, prior to the desired point in time. We perform this accounting as follows. As before, we first select only the log entries that refer to at least one player-controlled character. For each of the selected log entries (in chronological order), we retrieve the embedded vector for each of the entities that it describes (vectors v_m and v_n), and update each vector by taking a step (controlled by $\alpha \in [0, 1] \subset \mathcal{R}$) in the direction given by the other vector:

$$\begin{aligned} v_m &\leftarrow v_m + \alpha * v_n \\ v_n &\leftarrow v_n + \alpha * v_m \end{aligned} \tag{1}$$

Intuitively, entities that interact more become more aligned in the vector space. The step size, α, is one of the parameters that we varied in our experiments ($\alpha = 0.0025$, 0.05, or 0.10).

Our classification task is thus: Given an updated set of vectors that represents a particular point in time in a play session, predict whether two given entities will interact after that point in time, and before the play session ends. Training a classifier to perform this task requires labelled data, which we generate automatically using the $\sim$10% of gameplay sessions that we withheld from the graph building process. To provide ample data to the classifier, we use each of the play sessions to generate 100 labelled data points.

To generate a data point from a play session, we need a time point during that session, updated vectors for two entities at that time point, and a correct label concerning whether or not those entities interact after that point in time. We begin by choosing two entities at random from those that interact in the given session. Next, we search through the session's log entries to find the *final* interaction between the chosen entities. Finding the final interaction is useful because it allows us to choose our time point (before the final interaction, or after) in a way that guarantees the correctness of the label that we assign ("yes", or "no", respectively). The last step is to choose a specific time point and calculate updated vectors for our chosen entities, using the logs that precede that time point. Since we have some freedom in this choice (e.g., any point before the last interaction will do, for a "yes" answer), we constrain the choice in a way that reduces computation across the 100 generated samples.

1) Reduced Computation for Vector Updates: To avoid computing an updated set of vectors for each of the 100 samples that we generate from every play session, we first precompute a handful of updated vector sets for each play session, and then sample randomly from those vector sets when generating each data point. This works as follows.

For a given play session, we begin by segmenting it into adjacent slots of time. The logs of *Dota 2* offer a convenient basis for this segmentation, as they are already separated into adjacent periods of combat (called "team fights") and non-combat; for our experiments, we took each of these periods as a time slot. Short matches of *Dota 2* have few time slots, while a long match might have 20. For the last log entry in each of a session's slots, we compute and store an updated set of vectors, where every updated set accounts for all of the selected log entries that precede it, as explained in Equation 1. In Figure 3, each time slot is marked with T_k and each corresponding set of updated vectors is marked with V_k, where $k \geq 1$ is a number that identifies the slot/set.

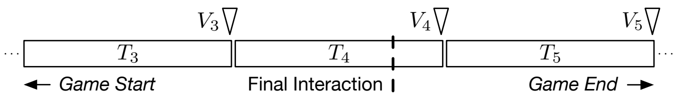


Fig. 3. A visualization of part of a play session, segmented into time slots. The Ts represent time slots and the Vs represent sets of updated vectors. The dashed line shows the last log entry in which entities m and n interacted.

We use the stored sets of vectors when generating each new classification sample. Given a chosen pair of entities (m, n), we find the final *time slot* in which the entities interact (T_{final}). For example, in Figure 3, T_{final} is T_4. To generate a sample with a "yes" label, we choose a random time slot earlier than T_{final} (if one exists) and we retrieve the updated vectors for m and n from those that were stored for the chosen time slot. In our example, we would randomly choose a time slot earlier than T_4, say, T_3, and then retrieve the updated vectors for m and n (v_m and v_n) from the set of vectors V_3. Our labelled sample would then consist of the tuple $\langle v_m, v_n, \text{yes}\rangle$. Generating a sample with a "no" label follows a similar process, but chooses a random time slot *after* T_{final} instead (if one exists). Given that this process can fail (for "yes" answers when T_{final} is the first time slot in the log, and for "no" answers when T_{final} is the last slot), we continue attempting to generate data points until 100 samples have been generated successfully. To generate a reasonably balanced data set, we generate "yes" and "no" answers each with 50% probability.

From our total dataset of 3281 *Dota 2* play sessions, we reserved 328 of them ($\sim$10%) for generating samples for classification. At 100 samples per play session, we thus generated 32800 samples to train and test each classifier.

C. *Training and Testing Classifiers*

To assess the quality of the vectors that are generated by a given embedding technique, we train and test a simple classifier using binomial logistic regression and 10-fold cross-validation. We posit that higher quality vectors can better inform the classifier, and thus achieve a higher accuracy on our prediction task. Nevertheless, it is important to recall that our goal is *not* to optimize any classifier for prediction accuracy. Instead, we seek to establish an indicator of vector quality that can be compared across embedding techniques and is straightforward to interpret.

We train and test a separate classifier for each embedding technique, each using a unique set of classification samples that we generated in the previous step. Finally, we compute each classifier's prediction accuracy. We use the resulting accuracies as indicators of how well each embedding technique was able to represent information about player behaviour in the embedded vectors that they produced. We take a higher accuracy to indicate a more useful set of embedded vectors – at least for predicting interactions between in-game entities.

V. Experiments, Results, and Analysis

We characterized the three embedding techniques that we tested by repeatedly applying our framework for experimentation across a range of different parameter configurations. The parameters that we varied were the *number of dimensions* of the embedded vector space, the *number of play sessions* used as input, the *types of interactions* used when building the graph, and the *step size* used when updating vectors during sample generation. To focus our analyses, we left any other parameters at their defaults, as given by the open-source implementations of the techniques that we used [26]–[28].

We varied the number of vector dimensions and the number of play sessions because they are both related to important practical tradeoffs. Vectors with fewer dimensions are cheaper to store and compute, but have less representational power. Meanwhile, minimizing the number of play sessions used can reduce computation and support games that have fewer logs available, but risks producing less meaningful vectors due to there being less information in the input. We varied the types of interactions that are used when building the graph to help understand how much the inclusion of each type affects the overall quality of the embedded vectors. Finally, we varied the step size for the vector updates to understand how each technique might respond to our simple updating scheme.

We tested four values for number of dimensions (16, 128, 256, 512), three values for number of games (100, 500, 3281), five ways to filter out types of interaction (leaving one of the following out: item purchases, kills, rune uses, ability uses, item uses), and three step sizes (0.0025, 0.05, 0.10). To simplify interpreting our results, we only varied one parameter at a time, using the following fixed values for the other three parameters: number of dimensions: 512, number of games: 3281, types of interactions: all, step size: 0.05. We settled on these "default" values via early informal testing, as they seemed to show clear differences across the three techniques. To enable fair comparisons, we used the same random seeds across all three techniques when splitting the play sessions, generating samples for classification, and splitting the samples for 10-fold cross-validation.

TABLE I
PREDICTION ACCURACIES FOR NUMBERS OF VECTOR DIMENSIONS.

	16	**128**	**256**	**512**
NODE2VEC	0.520	0.582	0.603	0.617
LINE	0.520	0.580	0.586	0.591
TGN	0.516	0.518	0.520	0.522

TABLE II
PREDICTION ACCURACIES FOR NUMBERS OF PLAY SESSIONS.

	100	**500**	**3281**
NODE2VEC	0.621	0.661	0.659
LINE	0.621	0.623	0.644
TGN	0.554	0.589	0.530

TABLE III
PREDICTION ACCURACIES WHEN ONE INTERACTION TYPE EXCLUDED.

	Purchase	**Kill**	**Rune**	**Ability**	**Item**
N2V	0.708	0.645	0.645	0.571	0.588
LINE	0.706	0.596	0.613	0.541	0.604
TGN	0.679	0.541	0.604	0.541	0.520

TABLE IV
PREDICTION ACCURACIES FOR STEP SIZES.

	0.0025	**0.05**	**0.10**
NODE2VEC	0.650	0.655	0.647
LINE	0.633	0.638	0.642
TGN	0.522	0.521	0.520

When each parameter value varies, the others remain fixed at these values: Dimensions: 512, Play Sessions: 3281, Interactions: All, Step Size: 0.05.

A. Results

Tables I to IV show the results of our experiments for each value of each parameter. The headings show the values of the parameter that we varied while collecting the data in the table; in Table III, the heading shows each type of player interaction that we excluded from the dataset.

Because of how we fixed values for the parameters that we held constant in each experiment, results for one specific parameter variation appear three times (num. dimensions: 512, num. games: 3281, all interactions included, step size: 0.05); these appear as the 512 column in Table I, the 3281 column in Table II, and the 0.05 column in Table IV. The reported accuracies are different because we varied the random seeds as described above before testing each range of parameters.

To help quantify the amount of variance that is inherent in the accuracies that we report (and thus help inform what differences can be considered meaningful), we collected more data for the parameter configuration that is represented in these three cases. Specifically, we split our 3281 play sessions into ten roughly equal segments (of 328 or 329 logs) and applied our experimentation framework once for each segment, treating it as the $\sim$10% of data to withhold and the remaining $\sim$90% as the data to use when building the graph. The average prediction accuracies across these ten experiments were (NODE2VEC: 0.623, LINE: 0.587, TGN: 0.526), and for all three techniques the standard deviation was less than 0.01. Though not conclusive, this suggests that differences in accuracy greater than 1-2% can be considered meaningful.

B. Analysis

As expected, the accuracies are not high – we made no attempts to optimize the simple predictors that we used, nor did we investigate whether any other predictors might result in stronger predictions. In our analysis, we are unconcerned with the *absolute* accuracy of any predictor, and very concerned with the *relative* differences in accuracies between multiple predictors. These relative differences help us understand how the embedding techniques that we tested fared at representing player behaviour, in comparison to (i) one another and (ii) themselves using different variations of their parameters.

From Table I, it seems clear that using only 16 vector dimensions was too few for both NODE2VEC and LINE, as the prediction accuracy due to their vectors jumped substantially when we used 128 dimensions or more. There may be diminishing returns for using more than 128 dimensions, however, as the gains in accuracy for doing so were minimal at best. TGN performed poorly regardless of how many dimensions we used, and it generally performed worse than both NODE2VEC and LINE across 9 of the 13 variations that we tested[1].

The Purchases column of Table III hints at an explanation for TGN's poor performance. Item purchases are peculiar as interactions in *Dota 2*, in that they tend to be heavily

[1]While Tables I to IV show a total of 15 values, we say that we tested only 13 variations because three of the 15 values all represent one specific variation of our parameters, as we explained in Section V-A.

overrepresented among all interactions and are often repeated both within and across play sessions (i.e., players buy many items, and they often buy a particular item multiple times). While the graph for NODE2VEC and LINE are unable to represent these repeat purchases (recall Section IV-A), the graph for TGN *does* represent all of these repeat purchases. Given that the representational power of the embedded vectors must be shared across all interaction types, we suspect that (i) the relative abundance of purchase interactions reduces the vectors' capacity to represent other types of interaction, and (ii) this has a particularly pronounced effect on TGN's vectors. The Purchases column of Table III supports these suspicions: when we removed the log entries related to item purchases before any graphs were built, the accuracies all improved to the highest that we observed across all of our experiments. Furthermore, the performance of TGN increased the most of all – from 0.530 (see the 3281 column of Table II) to 0.679.

The accuracies for TGN's vectors shown in Table II are also consistent with our suspicion about the effect of including purchase interactions when building TGN's graph. While training on more play sessions is helpful to a point (from 100 to 500 yields an improvement), training on all 3281 play sessions is worse than training on only 100. While the addition of more data from play sessions seems likely to improve accuracy (save overfitting), an increasing, damaging effect from the inclusion of more purchase interactions might overwhelm the benefit of more data, leading to the decrease in accuracy that we observed when using all 3281 play sessions.

As shown in Table IV, the different values that we tested for our vector update step size seem to have had no meaningful effect on the quality of the embedded vectors. Perhaps other values for step size might have a meaningful effect.

The performance of NODE2VEC is the strongest overall, obtaining higher accuracies than the other techniques in 7 of the 13 different variations that we tested.

In terms of storage efficiency, embedded vectors require substantially less storage than the logs that we used to compute the vectors. On average, each log file requires roughly 260KB of storage, amounting to 852.6MB for the 3281 logs that we used. In contrast, a set of roughly 1000, 512-dimensional vectors (one per graph node) stored with 64-bit precision requires only about 4.1MB – two orders of magnitude less than the log data that they aim to represent. This savings is particularly relevant in the context of game development, where the available space for volatile and non-volatile storage on entertainment consoles is highly constrained.

VI. Discussion and Future Work

The goal of our work was to characterize the performance of three vector embedding techniques with regard to their ability to represent player behaviour. We claim success – not because any of the models predicted particularly well (they did not) – but because our experiments revealed interesting, relative *differences* between the performances of the three techniques. Given the size of our dataset, we expect that these differences will persist across larger samples of *Dota 2* logs.

Looking beyond *Dota 2*, since our proposed method of building graphs from logs is general and avoids manual feature selection (i.e., collect all interactions between entities that interact), it simplifies the task of performing similar analyses of other games. Furthermore, while our proposed method of *testing* different embedding techniques requires log entries with attached timestamps, our method of *building* a graph from logs is free from this requirement. This further supports the use of embedding techniques across a wide range of games.

As we learned from our analysis, the embedding techniques appear to be sensitive to the relative abundance of different types of interaction, where the inclusion of a particularly abundant type of interaction can cause a degradation in the information that is represented about the other types. A potential improvement might be to compute a separate set of vectors for each type of interaction that is of interest in a game, as doing so would help prevent an abundant interaction type from causing valuable information about the other types to be excluded from the embedded vectors.

For those who seek to predict player behaviour using graph embedding techniques, the work that we have presented herein offers a general way to (i) build a suitable graph from game logs, (ii) generate training data and testing data for classification, and (iii) train and optimize classifiers that predict future interactions between game entities. Performing this optimization step was unnecessary for our work, as we only sought to find relative differences between prediction accuracies for the three embedding techniques that we tested. Nevertheless, our experimental framework supports such optimization directly: it would be straightforward to replace the simple classifier that we used with a more capable predictor, and the experimenter could then proceed with parameter tuning, using accuracies on our link prediction task as the target for optimization.

The characterization of NODE2VEC, LINE, and TGN that we present in this paper could depend on our method for building graphs from gameplay logs. While this method has some desirable properties, it would be interesting to explore how other methods of building a graph might affect the relative performance of these or other embedding techniques.

The embedding techniques that we tested vary with respect to the kinds of graph information that they aim to represent: NODE2VEC (with parameters $p = q = 1$) aims to represent the graph's structure, LINE aims to represent neighbour relationships, and TGN aims to represent changes to edges over time. It would be interesting to investigate whether any of these aims are particularly well or poorly suited to embedding graphs that represent interactions between game entities.

While the link prediction task that we used in this paper offers one way to measure the performance of different graph embedding techniques, it could be useful to develop metrics that cover other desirable properties of player models.

Finally, given that interactions between game entities are unlikely to capture properties of players that would be useful for personalization, it would also be valuable to study how models built using our method might be integrated with other sources of player data.

VII. Conclusion

To the best of our knowledge, this paper represents the first application and investigation of graph embedding techniques for representing player behaviour in computer games. In carrying out this work, we made the following **contributions**. First, we devised a general way to build a graph based on logs of player interactions, which we then applied to a set of more than 3000 logs from the popular computer game, *Dota 2*. Our method is straightforward to apply to any logs that record player interactions with game entities, and it avoids any need to manually select features. Second, we adapted the link prediction task from prior work to suit the context of player modelling, and we proposed an efficient way to generate the required samples for classification. This work will directly support future efforts to study graph embedding techniques as a way to represent player behaviour, as it presents a clear task to attempt and offers a tractable way to pursue it. Third, we characterized the behaviour of open-source implementations of three embedding techniques using data from a well-known commercial game. Not only does this provide data that can be used for comparison as other embedding techniques are investigated, but it helps practitioners understand the relative strengths of NODE2VEC, LINE, and TGN in a familiar and relevant context. Finally, we offer the complete source code of our experimental framework as a downloadable, open-source project, to support both replication and further evaluations [29]. Having a compact, high-quality representation of player behaviour would be a substantial boon to personalization in games, and we offer this research as an early step forward in pursuit of that goal.

Acknowledgements

We are grateful to the authors of the OpenDota API [20], as well as the developers of the NODE2VEC, LINE, and TGN implementations that we used in this work [26]–[28]. We also acknowledge the support of the Natural Sciences and Engineering Research Council of Canada (NSERC), Grant #2020-06502, and the Mitacs Research Training Award. Some parts of this work appear in the lead author's dissertation [30].

References

[1] D. H. Kwak, G. E. Clavio, A. N. Eagleman, and K. T. Kim, "Exploring the antecedents and consequences of personalizing sport video game experiences," *Sport Marketing Quart.*, vol. 19, no. 4, pp. 217–225, 2010.

[2] D. Thue, V. Bulitko, M. Spetch, and E. Wasylishen, "Interactive storytelling: A player modelling approach," in *3rd AI and Interactive Digital Entertainment Conference (AIIDE 2007)*. AAAI Press, 2007, pp. 43–48.

[3] S. Snodgrass, O. Mohaddesi, and C. Harteveld, "Towards a generalized player model through the PEAS framework," in *Proceedings of the 14th International Conference on the Foundations of Digital Games*, ser. FDG '19. New York, NY, USA: ACM, 2019.

[4] C. Eggert, M. Herrlich, J. Smeddinck, and R. Malaka, "Classification of player roles in the team-based multi-player game dota 2," in *Entertainment Computing - ICEC 2015*, K. Chorianopoulos, M. Divitini, J. Baalsrud Hauge, L. Jaccheri, and R. Malaka, Eds., vol. LNCS 9353. Springer International Publishing, 2015, pp. 112–125.

[5] J. Pirker, S. Griesmayr, A. Drachen, and R. Sifa, "How playstyles evolve: Progression analysis and profiling in just cause 2," in *Entertainment Computing - ICEC 2016*, G. Wallner, S. Kriglstein, H. Hlavacs, R. Malaka, A. Lugmayr, and H.-S. Yang, Eds., vol. LNCS 9926. Springer International Publishing, 2016, pp. 90–101.

[6] Valve Corporation, "Defense of the Ancients 2," www.dota2.com, 2013.

[7] N. Y. Khameneh and M. Guzdial, "Entity embedding as game representation," in *Proceedings of the AIIDE Workshop on Experimental AI in Games (EXAG)*, vol. 2862. CEUR-WS, 2020, p. 7 pages.

[8] K. Makantasis, A. Liapis, and G. N. Yannakakis, "The pixels and sounds of emotion: General-purpose representations of arousal in games," *IEEE Transactions on Affective Computing*, 2021.

[9] T. Mikolov, I. Sutskever, K. Chen, G. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," in *Proceedings of the 26th International Conference on Neural Information Processing Systems - Volume 2*, ser. NIPS'13. Red Hook, NY, USA: Curran Associates Inc., 2013, pp. 3111–3119.

[10] I. Mordatch, "Concept learning with energy-based models," *arXiv preprint*, no. 1811.02486, p. 12 pages, 2018.

[11] ——, "Concept learning with energy-based models," in *Proceedings of the 6th International Conference on Learning Representations (ICLR 2018), Workshop Track*. OpenReview.net, 2018, p. 6 pages.

[12] Y. Du, S. Li, and I. Mordatch, "Compositional visual generation with energy based models," in *Advances in Neural Information Processing Systems*, H. Larochelle, M. Ranzato, R. Hadsell, M. Balcan, and H. Lin, Eds., vol. 33. Curran Associates, Inc., 2020, pp. 6637–6647.

[13] A. Grover and J. Leskovec, "node2vec: Scalable feature learning for networks," in *Proceedings of the 22nd ACM SIGKDD international conference on Knowledge discovery and data mining*. New York, NY, USA: Association for Computing Machinery, 2016, pp. 855–864.

[14] J. Tang, M. Qu, M. Wang, M. Zhang, J. Yan, and Q. Mei, "LINE: Large-scale information network embedding," in *Proceedings of the 24th International Conference on World Wide Web*. New York, NY, USA: Association for Computing Machinery, 2015, pp. 1067–1077.

[15] E. Rossi, B. Chamberlain, F. Frasca, D. Eynard, F. Monti, and M. Bronstein, "Temporal graph networks for deep learning on dynamic graphs," in *ICML 2020 Workshop on Graph Representation Learning and Beyond*, no. 2006.10637. arXiv, 2020.

[16] D. Xu, C. Ruan, E. Korpeoglu, S. Kumar, and K. Achan, "Inductive representation learning on temporal graphs," in *Proceedings of the 8th International Conference on Learning Representations*. OpenReview.net, 2020. [Online]. Available: https://openreview.net/forum?id=rJeW1yHYwH

[17] PUBG Studios, "PlayerUnknown's Battlegrounds," pubg.com, 2017.

[18] Blizzard Entertainment, "StarCraft II," starcraft2.com, 2010.

[19] Riot Games, "League of Legends," leagueoflegends.com, 2009.

[20] A. Cui, H. Chung, and N. Hanson-Holtry, "OpenDota," opendota.com, 2014.

[21] M. Guzdial, B. Li, and M. O. Riedl, "Game engine learning from video," in *Proceedings of the 26th International Joint Conference on Artificial Intelligence*, ser. IJCAI'17. AAAI Press, 2017, pp. 3707–3713.

[22] Avalanche Studios, "Just Cause 2," https://avalanchestudios.com/games/just-cause-2, March 2010.

[23] Y. Rabii and M. Cook, "Revealing game dynamics via word embeddings of gameplay data," in *Proceedings of the 7th AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, ser. AIIDE'21. AAAI Press, 2021, pp. 187–194.

[24] S. P. Singh, A. Kumar, H. Darbari, L. Singh, A. Rastogi, and S. Jain, "Machine translation using deep learning: An overview," in *2017 International Conference on Computer, Communications and Electronics (Comptelix)*. IEEE, 2017, pp. 162–167.

[25] Z. Li, Z. Cui, S. Wu, X. Zhang, and L. Wang, "Fi-GNN: Modeling feature interactions via graph neural networks for ctr prediction," in *Proceedings of the 28th ACM International Conference on Information and Knowledge Management*, ser. CIKM '19. New York, NY, USA: Association for Computing Machinery, 2019, pp. 539–548.

[26] A. Grover, "node2vec source code," https://github.com/aditya-grover/node2vec, January 2017.

[27] V. Noroozi, "LINE source code," https://github.com/VahidooX/LINE, January 2017.

[28] Twitter Research, "TGN source code," https://github.com/twitter-research/tgn, December 2020.

[29] J. Shah, "Behaviours as vectors," https://github.com/jinayshah7/behaviours-as-vectors-thesis-code, May 2023.

[30] ——, "Vector embedding techniques for player behaviour in Dota 2," Master's thesis, School of IT, Carleton University, 2021.