DEGREE PROJECT IN MATHEMATICS,
SECOND CYCLE, 30 CREDITS
STOCKHOLM, SWEDEN 2019

Towards Machine Learning Enabled Automatic Design of IT-Network Architectures

LOVA WÅHLIN

KTH ROYAL INSTITUTE OF TECHNOLOGY
SCHOOL OF ENGINEERING SCIENCES

Towards Machine Learning Enabled Automatic Design of IT-Network Architectures

LOVA WÅHLIN

Degree Projects in Mathematical Statistics (30 ECTS credits)
Degree Programme in Applied and Computational Mathematics (120 credits)
KTH Royal Institute of Technology year 2019
Supervisor at foreseeti AB: Per Eliasson
Supervisor at KTH: Pontus Johnson
Examiner at KTH: Jimmy Olssoni

TRITA-SCI-GRU 2019:046
MAT-E 2019:16

Royal Institute of Technology
School of Engineering Sciences
KTH SCI
SE-100 44 Stockholm, Sweden
URL: www.kth.se/sci

Abstract

There are many machine learning techniques that cannot be performed on graph-data. Techniques such as graph embedding, i.e mapping a graph to a vector, can open up a variety of machine learning solutions. This thesis addresses to what extent static graph embedding techniques can capture important characteristics of an IT-architecture graph, with the purpose of embedding the graphs in a common euclidean vector space that can serve as the state space in a reinforcement learning setup. The metric used for evaluating the performance of the embedding is the security of the graph, i.e the time it would take for an unauthorized attacker to penetrate the IT-architecture graph. The algorithms evaluated in this work are the node embedding methods node2vec and gat2vec and the graph embedding method graph2vec. The predictive results of the embeddings are compared with two baseline methods. The results of each of the algorithms mostly display a significant predictive performance improvement compared to the baseline, where the F_1 score in some cases is doubled. Indeed, the results indicate that static graph embedding methods can in fact capture some information about the security of an IT-architecture. However, no conclusion can be made whether a static graph embedding is actually the best contender for posing as the state space in a reinforcement learning framework. To make a certain conclusion other options has to be researched, such as dynamic graph embedding methods.

Keywords: IT-Architecture graph, Node Embedding, Graph Embedding, Reinforcement Learning, Machine Learning

Sammanfattning

Det är många maskininlärningtekniker som inte kan appliceras på data i form av en graf. Tekniker som graph embedding, med andra ord att mappa en graf till ett vektorrum, can öppna upp för en större variation av maskininlärningslösningar. Det här examensarbetet evaluerar hur väl statiska graph embeddings kan fånga viktiga säkerhetsegenkaper hos en IT-arkitektur som är modellerad som en graf, med syftet att användas i en reinforcement learning algoritm. Dom egenskaper i grafen som används för att validera embedding metoderna är hur lång tid det skulle ta för en obehörig attackerare att penetrera IT-arkitekturen. Algorithmerna som implementeras är node embedding metoderna node2vec och gat2vec, samt graph embedding metoden graph2vec. Dom prediktiva resultaten är jämförda med två basmetoder. Resultaten av alla tre medoterna visar tydliga förbättringar relativt basmetoderna, där F_1 värden i några fall uppvisar en fördubbling. Det går alltså att dra slutsatsen att att alla tre metoder kan fånga upp säkerhetsegenskaper i en IT-arkitektur. Dock går det inte att säga att statiska graph embeddings är den bästa lösningen till att representera en graf i en reinforcement learning algoritm, det finns andra komplikationer med statiska metoder, till exempel att embeddings från dessa metoder inte kan generaliseras till data som inte var använd till träning. För att kunna dra en absolut slutsats krävs mer undersökning, till exempel av dynamiska graph embedding metoder.

Nyckelord: IT-Arkitektur, Node Embedding, Graph Embedding, Reinforcement learning, Maskininlärning

Acknowledgements

I would like to reach out to all the people who helped and supported me during this project. First and foremost to everyone at foreseeti that made each day in the office enjoyable. I am very grateful for the wonderful workplace that I had and the desk that I knew would be reserved for me even after some travel adventures that would leave me absent for a few days. I want to thank **Pontus Johnson**, my supervisor at KTH, for guiding me throughout the project and always being kind, helpful and encouraging. I want to thank **Per Eliasson**, my supervisor at foreseeti, for always making sure that I had a good working environment in the office, discussing various concept of the simulator with me and helping me with all kinds of computer related questions. I would also like to thank **Jo Tryti**, my early companion in the project, for introducing me to our task, guiding me through the big forest of code and discussing various solutions with me. My sincerest gratitude to my mom **Ingrid Claesson** and **Jarl Sobel** for reviewing and giving me feedback on the report and for listening and being so very kind and supportive throughout all ups and downs. I would like to reach out to **Zacharias Fisches** for giving me such great ideas, for spending hours to discuss obstacles with me and for being an enormous support during this whole project, it would have been so much harder without you. Lastly, I want to thank my dad **Christian Wåhlin** for inspiring and encouraging me to become an engineer, I miss you immensely and I wish I could share this experience with you.

Contents

Chapter 1

Introduction

The world is more connected now than ever before, and the connections are growing each day as the technology moves forward. Today's globally interconnected infrastructures give freedom while at the same time demanding more up-to-date security. All companies today in the tech industry have data that is of high importance, and it would have devastating consequences for the company or for individuals if it was compromised. Moreover, as the security knowledge of defenders of the systems increase, so does the knowledge of the attackers. Consequently, the IT-security systems need to be advanced and constantly updated and improved, since a secure system today might not be secure tomorrow. This constantly changing environment can be confusing and hard to understand for any individual who does not possess expert knowledge in the area.

The company foreseeti addresses the problem of making IT-security understandable, not only for the experts but for everyone. They do it by visualizing the IT-security architecture of a company in a graph, such that it clearly displays the IT-structure of the company. They developed the tool securiCAD, that allows their customers to draw their IT-architecture in a CAD-like manner. Furthermore, it implements Bayesian networks [19] to model the risk exposure of the IT-architecture using probabilistic methods. SecuriCAD provides a "heat map" of where the architecture is more or less likely to be vulnerable to attacks, giving the costumers a clear and continuous risk assessment while they plan and design future architectures.

In a *work in progress* paper [22] by Lagerström et al. the development of an *Automatic Designer* within the securiCAD tool is suggested. The Automatic Designer employs machine learning techniques to automatically improve and redesign any IT-architecture modeled in securiCAD. Consequently, any customers using the securiCAD tool would not only get the visualization and risk assessment of the current architecture, but they would also get suggestions of how to remove, add and modify components to improve it on the spot. The approach suggested for implementing the Automatic Designer is reinforcement learning (RL), where an agent is trained to make modifications on the existing IT-architecture to improve it. The current state of the agent is the current IT-architecture design. In the implementation of the *Automatic Designer* the state space is a finitely

large set of IT-architecture designs created using the securiCAD tool. Consequently, each of these designs is in the form of a graph, with vertices connected by edges. Each of these vertices and edges represents a real entity or connection in an IT-architecture system. Consequently, they employ all the imaginable attributes required to model an IT-architecture realistically, which is a significant amount. As a result, the state space in the implementation of the Automatic Designer is in fact a set of heavily attributed graphs.

As turns out, many machine learning techniques cannot be applied directly to graph data [23], which is likewise the case for the RL setup in the *work in progress* paper. Graphs solely consists of a set of vertices and a set of edges, and it is a challenge to represent this structure so that it can easily be exploited by machine learning models [16]. As far as the knowledge of the author of this thesis goes, there has not been any publications of any attempts to perform RL on graph data in the way that the Automatic Designer suggests. However, *learning representations* of data has become a field in itself in the machine learning community [3]. Learning a representation means to transform the data to make it more accessible and easier to extract useful information from. This field has shown great success in e.g speech recognition, object recognition and natural language processing. Recent methods propose representation learning on graphs, where the nodes (vertices), subgraphs or entire graphs are mapped into an euclidean vector space. These algorithms are referred to as *graph embedding* or *node embedding* methods. The idea of an embedding is that two nodes that are embedded closely in this vector space, are also similar to each other in the graph according to some *similarity measure*. How this similarity measure of two nodes in a graph is defined differs for different embedding algorithms. In conclusion, a vector space is fit for a richer tool set of machine learning approaches than graph structures, including RL.

For the implementation of the Automatic Designer, a graph embedding method could be a viable solution for representing the attributed graphs in the state space of the RL setup. There are many different areas of research regarding graph embedding. Most algorithms are *static*, i.e they are applicable to graphs that do not evolve over time [23]. Thus, in the framework of the Automatic Designer, each individual graph in the state space could be represented by one or several vectors in some euclidean vector space using a static graph embedding.

This thesis regards this topic, and investigates how a static graph embedding can pose as the state space in a reinforcement learning framework. There are several ways to evaluate whether or not an embedding is truly representative of a graph. In literature, it is common to evaluate an embedding by performing node and edge prediction, i.e trying to predict the class belonging of the nodes and edges in the graph given the embedding. If successful, the embedding is assumed to have captured some important characteristics of the graph. In the Automatic Designer, it would be desirable if the embedding captured characteristics of how *secure* the graph is, since that would be an indicator whether the graph has to be modified or not. In securiCAD, there is more than one way to measure security. However, the most used measure is the *Time-To-Compromise* (TTC), which indicates how long it would take for an unauthorized attacker to compromise a specific node. If this number is low, it is easy to compromise the node and thus the security is low. In summary, if the TTC

values can be predicted given some static graph embedding, it is an indicator that these techniques can be useful when implementing the Automatic designer.

For investigating this topic a *simulator* of the securiCAD tool is at disposal. The simulator employs all the major functionalities of securiCAD. However, it is a simplified version and thus does not incorporate all the attributes of the securiCAD. Nontheless, it can build IT-security graphs, simulate attacks and calculate TTC values. The assumption is that if the TTC values can be predicted using static graph embeddings in the simulator, it will show similar results when expanded to the more advanced securiCAD.

1.1 Previous Work

There are many different research areas regarding static graph representation. Most of them regard embedding individual nodes, i.e each node in the graph is mapped into a vector. These methods are generally optimized for performing link and node prediction, and they are trained in an unsupervised or semi-supervised manner. The static node embedding methods can roughly be divided into three categories [16], matrix-factorization methods [1, 2, 5, 38], random walk based methods [14, 39, 41, 43] and deep learning methods [6, 29, 47].

The matrix factorization methods represent the connections between nodes in a graph as some matrix, and applies a factorization to it to obtain a lower dimensional representation of it. One of the earliest factorization method is Laplacian Eigenmaps [2] from 2002. The matrix used for factorization is commonly the adjacency matrix. A power of the adjacency matrix is employed by Cao et al. in Grarep [5]. Another algorithm is HOPE [38] that captures directed edges and uses generalized singular value decomposition (SVD) to map the matrix representation to lower dimensions.

The random walk methods perform random walks from each node in the graph to obtain a neighborhood set that can be used to train the vector representation of that node. The assumption is that two nodes that co-occur often during random walks over the network should have similar node representations. What mainly differs these methods is how they define their random walks. The DeepWalk [41] algorithm uses a fixed length random walk, whereas node2vec [14] uses a *biased* random walk strategy to trade off between both local and global views of the network. An extension to the random walk strategies is HARP [7], that uses a pre-processing step that collapses related nodes into ”supernodes” to improve the performance of the random walk methods.

The deep learning methods for graph representation incorporate deep neural network layers to obtain node embeddings. SEANO [29] uses a two-layer neural network, and DNGR [6] and SDNE [47] use autoencoders to compress information about the neighborhood of a node. There are methods that belong to more than one category, such as S2S-N2N-PP [45]. It combines the random walk strategies with the deep learning methods.

Apart from the static node embedding methods, there are methods that represent neighborhoods

or entire graphs as vectors. One such method is Subgraph2vec [35] by Narayan et al. that embeds smaller neighborhoods, and graph2vec [36] that embeds whole graphs.

1.2 Research Questions

The main purpose of this thesis is to investigate one specific approach for representing graph-data as the state space in the reinforcement learning setup in the *Automatic Designer* [22]. This approach is to map each graph to a vector using static graph and node embedding techniques. The questions to answer in this thesis are:

- How well can a static node embedding capture important characteristics of an IT-security graph?
- How well can a static graph embedding algorithm capture important characteristics of an IT-security graph?
- How viable is the approach of using a static graph/node embedding for the purpose of representing a graph as the state space in a reinforcement learning setup?

1.3 Outline

The outline of this thesis is as follows. The background consists of chapter 2 and chapter 3, where the former describes the securiCAD and the simulator, and the latter theory about graph and node embedding. Subsequently, the methodology is in chapter 4 followed by results in chapter 5. Thereafter, a chapter about potential future work and discussion in chapter 6, and lastly a short conclusion in chapter 7.

1.4 Nomenclature

AP	Attacker Persistence
BFS	Breadth-First Sampling
DFS	Depth-First Sampling
DTR	Decision Tree Regression
RL	Reinforcement Learning
SGD	Stochastic Gradient Descent
SVC	Support Vector Classifier
SVM	Support Vector Machine
TTC	Time To Compromise
UNI	User Needs Information

Chapter 2

SecuriCAD and the Simulator

The company foreseeti enables firms to conduct threat modeling and cyber risk simulation through their product securiCAD. The tool lets companies draw their IT-architecture in a CAD-like manner in the structure of a graph and simulates hacker attacks to provide a ”heat map” [20] on where the architecture is more or less vulnerable. It assesses the time it would take for a professional attacker to compromise a specific asset. It is a complex tool, and it employs numerous features. For conducting the experiments in this thesis work, a simplified simulator of the SecuriCAD is employed. For this reason, the securiCAD is only briefly described in the chapter 2.1 and for further details the reader is referred to [20] and [10]. The simulator is described in more detail in chapter 2.2.

2.1 SecurCAD

The SecuriLang [20] has four types of concepts; Attacker, AttackStep, Defence, and Asset.

Asset The assets are the vertices in the graph created by the customer, by drawing the architecture using securiCAD. Each asset represents a logical, physical or conceptual part of an IT-architecture. Examples of assets are: Firewall, Host, Intrusion Detection System (IDS), Network, Protocol, Router. The edges in the graph represent different system relation types between the assets.

Defence Most assets are equipped with defences. Each defence reduces the vulnerability of the asset and prolongs the time it takes for an attacker to compromise it. Examples of defences associated with the asset Host are: AntiMalware, Hardened, HostFirewall, Patched, StaticARPTables.

Attacker An attacker is an individual who wants to gain access to assets with or without proper authority, to exploit weaknesses or cause damage. In SecuriCAD it is assumed that the attacker is a professional penetration tester that has all public knowledge and all possible tools. An Attacker is likewise an asset, i.e modelled as a vertex in the graph of the IT-architecture.

AttackStep An attack step is a step the attacker makes to penetrate further into the IT-architecture. Each attack step is connected to an asset and the defences that protect it. Every attack step has a set of parent and child attack steps. A parent-child relationship is represented by an AttackStepMin (or-step) or an AttackStepMax (and-step). The former implies that *at least one* of the parent attack steps must be successfully taken for it to be taken, and the latter that *all* of the parents must be successfully taken. Examples of attacks on the asset Host are: BypassAntiMalware, BypassIDS, Compromise, PhysicalAccess, UserAccess.

Given an IT-security graph drawn in securiCAD, an attack graph can be generated by simulating cyber attacks on it. The attack graph consists of vertices modelling the attack steps from the entry point of the attacker and edges modelling child/parent relationships between the attack steps. The attack steps are modelled in a probabilistic fashion, no nodes are considered safe but the probability of compromise may be very small.

2.1.1 Time-To-Compromise (TTC)

To have a measure of the effort expected of the attacker to move through the attack graph and compromise assets, the measure Time-To-Compromise (TTC) is used. The TTC for each attack step, i.e for reaching a new node in the attack graph, is modelled in a probabilistic fashion as a random variable and thus connected to some probability distribution. With the assumption that the effort of the attacker is expended uniformly, the TTC for each step can be calculated through repeated random sampling from the probability distribution. By the Monte Carlo method the TTC estimate is the sample mean:

$$TTC^i = \sum_{j=0}^{N} \frac{f(x_j^i)}{N}$$

where x^i is a sample drawn from the probability distribution associated with attack step i and N the number of samples, $f(\cdot)$ is an optional weight function. SecuriCAD employs five cumulative distribution functions [20], Bernoulli, Exponential, Gamma, Log-normal and Pareto. To find the shortest path, i.e the lowest TTC value to each node, can be fit into the framework of a classical network optimization problem. The algorithm used in SecuriCAD is *Dial's Approximate Buckets* with a description found in [8].

2.2 The Simulator

The simulator is considerably less complex than SecuriCAD, but attempts to capture the most important aspects of SecuriCAD. As far as the knowledge of the author of this thesis goes, there is no documentation of the simulator except of what is written in this report. Therefore, the description in this chapter is done with the intention to correctly describe how the simulator works. However, it is not to be taken for true documentation, but can give an introduction.

2.2.1 Assets

There are seven assets in the simulator; Host, Network, Dataflow, Credentials, Information, Attacker and User. An overview of their functionality is found in the table below.

Assets in the simulator		
Host	(H)	A computer or some kind of hardware that can store information.
Network	(N)	Connected to hosts, can be interpreted as a router.
Dataflow	(DF)	Data that is transferred from one host to another.
Credentials	(C)	Authorization required to legitimately access one host from another host.
Information	(I)	Important data stored on a host.
Attacker	(A)	An infiltrator that attacks the system to compromise the information.
User	(U)	A legitimate user of the system. Needs access to the information.

Except for the attacker and the user, there can be unlimited number of each asset in one IT-architecture graph. Most assets are connected to hosts with a *connect* connection. The exception are the credentials that will have either a *store* connection or a *grant access* connection. This induces that the authorization key/password that grants access to one host is stored on the other, and if the latter is reached by the attacker the former will likewise be easily accessed. The dataflow needs to be connected to *exactly* two hosts. Further, there is a value called *user needs information* (UNI) connected to a host and to one or more informations, which is an indicator of the importance of the user's accessibility of that information. However, the UNI is not a structural connection, and the attacker cannot exploit this connection to reach the information. There is an overview of the allowed connections between the assets in table 2.1.

Table 2.1: *The allowed connections in the Simulator: 1=Connect, 2=Store, 3=Grant access, –=No allowed connection. UNI stands for* User needs information *and is an indicator of the importance of the user's accessibility of that information.*

	H	N	D	C	I	A	U
Host	–	1	1 *	2, 3	1	1	1
Network	1	1 **	–	–	–	–	–
Dataflow	1 *	–	–	–	–	–	–
Credentials	2, 3	–	–	–	–	–	–
Information	1	–	–	–	–	–	UNI
Attacker	1	–	–	–	–	–	–
User	1	–	–	–	UNI	–	–

** Dataflow needs to be connected to* exactly *two hosts*
*** Network-Network connection implements a firewall*

All assets are connected to hosts, and hosts can be connected to anything but themselves. The network asset is the only asset that can be connected to itself, in which case a firewall is implemented, thus making it harder for an attacker to pass through.

Figure 2.1 shows an example of a very simple IT-architecture with all of the seven assets. Both the attacker and the user need to get from host 1 (H1) to host 2 (H2) in order to reach the information.

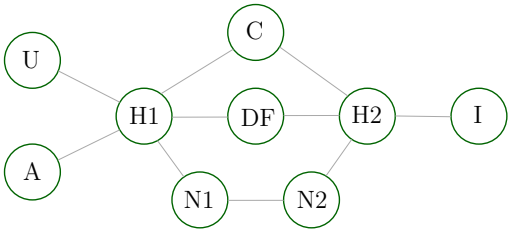

Figure 2.1: *An example of a simple graph of an IT-architecture in the simulator. U=User, A=Attacker, H1=Host number 1, H2=Host number 2, N1=Network number 1, N2=Network number 2, DF=Dataflow, I=Information.*

2.2.2 Attack Graph

Similarly to the SecuriLang language the simulator generates an attack graph starting from the attacker, and each node represents one attack step whose local TTC value is equipped with a probability distribution. However, contrary to SecuriLang the simulator exclusively employs the gamma distribution $X \sim \Gamma(k, \theta)$, with probability density function, mean and variance:

$$\begin{aligned} f(x; k, \theta) &= \frac{x^{k-1} e^{-\frac{x}{\theta}}}{\theta^k \Gamma(k)}, \quad x, k, \theta > 0 \\ E[X] &= k\theta \\ Var(X) &= k\theta^2 \end{aligned} \tag{2.1}$$

using the shape and scale parametrization, k and θ. In the simulator the attacks are divided into *legitimate* and *illegitimate* attack steps, where the expected TTC value $E[X] = k\theta$ is higher for an illegitimate attack step. The reasoning is that it takes a longer time for an attacker to access an asset by exploit than to do it by using for instance credentials.

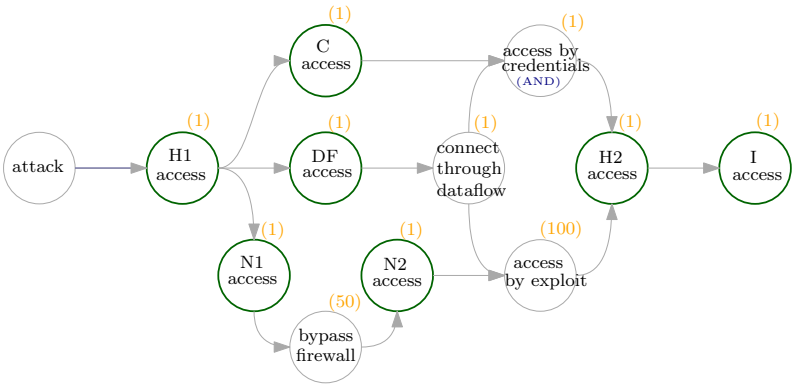

Figure 2.2: *An attack graph generated in simulator of the simple IT-architecture in figure 2.1. The legitimate attack steps are cheap, in this example it costs 1 TTC to take them, whereas the illegitimate ones,* bypass firewall *and* access exploit*, cost 50 and 100 respectively. Access by credentials is an (AND) step, which means that both parents,* C access *and* connect through dataflow*, have to be reached to get there. The attacker takes the cheapest path available.*

Figure 2.2 displays an attack graph of the small IT architecture in figure 2.1. The numbers above each node are the TTC values for that attack step. Note that these numbers are drawn and averaged from a probability distribution, thus the numbers in the figure are examples. The node

access by credentials is an (AND) attack step, which indicates that both its parents need to be compromised for it to be reached. The parents are *connect through dataflow* and *C access*. The attacker will choose the cheapest path to the information, hence through *H1 access*, *C access*, *access by credentials* and *H2 access* which leads to *I access*. This path costs the attacker 5 TTC.

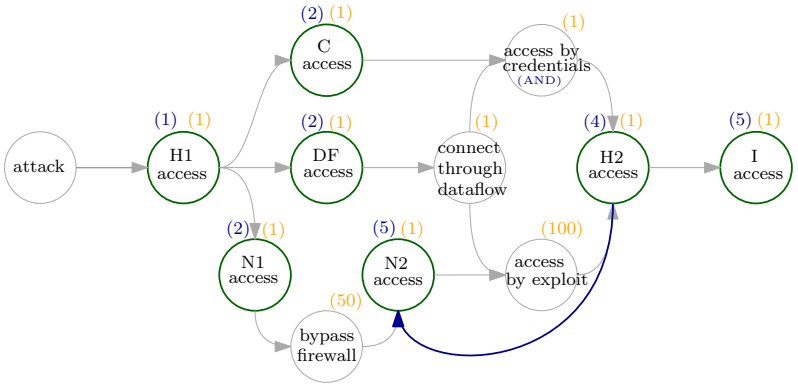

Figure 2.3: *The same attack graph as figure 2.2 showing the TTC value for each attack step in orange as well as the total TTC value for the attacker to reach each asset in blue. Since H2 and N2 are structurally connected (see figure 2.1), the attacker can in fact reach N2 without passing through the firewall.*

Figure 2.3 displays the very same attack graph as figure 2.2, while it also displays the total TTC for all the nodes in the actual IT-architecture graph in blue. The shortest path for the attacker to N2 is via N1. However, that includes the illegitimate attackstep *bypass firewall* and that path would costs 53 TTC. The attacker only pays 4 TTC to compromise H2, and H2 is structurally connected to N2, hence this path is cheaper since the attacker does not need to go through *bypass firewall*. See figure 2.1, access to H2 naturally leads to access of N2.

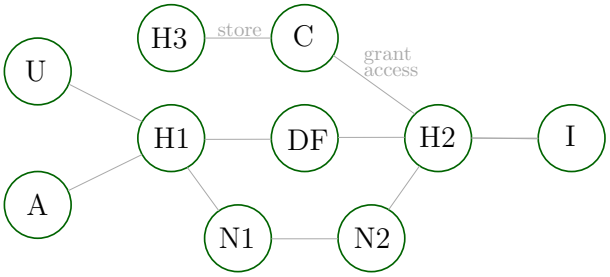

Figure 2.4: *An example of another IT architecture. Here the credential that grants access to host 2 are stored on host 3, which the attacker cannot reach.*

Figure 2.4 displays a similar IT-architecture to the example in figure 2.1, with the introduction of a third host H3 that stores the credentials instead of host H1. The corresponding attack graph is displayed in figure 2.5. In this case the attacker cannot reach the (AND) attack step *access by credentials*, since the credentials are not stored on H1 and the attacker cannot reach H3. Consequently, the total TTC of *C access* is infinity and the attacker needs to take the path through *access by exploit*. In this example the total TTC for *I access* is instead 105.

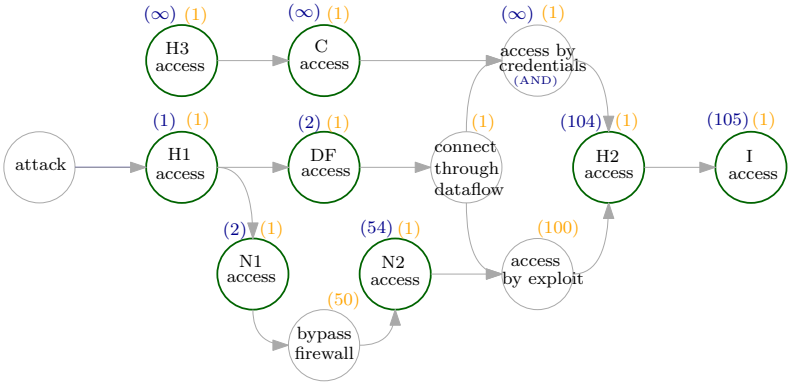

Figure 2.5: *The attack graph generated from the IT architecture in figure 2.4. Since the attacker can not reach both parents of* access by credentials*, the shortest way to the information is to* connect through dataflow *and* access by exploit*. In this example the cheapest path to* I access *costs 105 TTC.*

2.2.3 Total Cost

There are more measures than the TTC values that dictate how good an IT-architecture is. For an architecture design it is naturally important that the attacker can not easily compromise important information, however it is likewise important that the users have access to it. There are more parameters to the simulator that will not be described in detail, since they are not relevant for conducting the experiments in this thesis. These parameters are briefly mentioned for completeness in the table.

Parameters in the simualor		
Attacker persistence	AP	Skill of the attacker
User needs information	UNI	A measure of the importance of accessing the information
Unit cost	UC	A penalty cost for having many entities in a graph
Cost of compromise	COC	The cost of the damages if an information is compromised

The total cost that the simulator calculates is a function of these parameters, along with the attacker's TTC values and how long time it takes for the user to access the information it needs. Nonetheless, the sole focus in this thesis are the TTC values of the attacker, and further description of the rest of the parameters is omitted.

Chapter 3

Graph Embedding

This chapter will introduce static graph embeddings and describe the details about the algorithms implemented in thesis. The term *graph embedding* is used to describe a representation of each node of the graph, a sub-graph or an entire graph in a euclidean vector space. The embedding should capture the graph's structure, connections between individual nodes and its attributes. The intuition is to train a mapping of nodes/sub-graphs/graphs to d-dimensional vectors, such that nodes/sub-graphs/graphs that are similar to each other according to some proximity, are likewise close to each other in this d-dimensional vector space [25]. The proximity measure is defined differently for each algorithm, and depends on what characteristics the user wish to capture. A formal definition of a node embedding is found in definition 5.

In this thesis node embeddings and graph embeddings are further discussed in chapter 3.3 and 3.4. Sub-graph embeddings are not implemented and further description in this thesis is omitted.

3.1 Formal Definitions

Graphs are used as an efficient way to represent data and characterize actions between objects of interest. In this work a definition of a graph similar to the definition in [47] is used:

> **Definition 1** (Graph)**.** *A graph is denoted as $G = (V, E)$, where V=$\{v_1,\ldots,v_n\}$ is a set of n vertices (a.k.a nodes) and $E = \{e_{i,j}\}_{i,j=1}^{n}$ is a set of edges. Each edge $e_{i,j}$ is associated with a weight $s_{i,j} \geq 0$. If v_i and v_j are not connected to each other, then $s_{i,j} = 0$. Otherwise, for unweighted graphs $s_{i,j} = 1$ and for weighted graphs $s_{i,j} > 0$. For undirected weighted graphs, $s_{ij} = s_{ji}$ $\forall i, j \in \{1, \ldots, n\}$.*

All IT-architecture graphs generated from the simulator (chapter 2.2) are undirected and unweighted. Consequently,

$$s_{i,j} = s_{j,i} = 1 \quad \forall s_{i,j} \neq 0 \quad i, j \in \{1, \ldots, n\}.$$

A graph is often represented with a neighborhood matrix, also called adjacency matrix. The adjacency matrix displays the connections between vertices and is oftentimes used in eg. factorization based methods, chapter 3.3.2. The following definition of an adjacency matrix is similar to one by [5].

Definition 2 (Adjacency Matrix)**.** *An adjacency matrix of an unweighted graph is a matrix S of size $n \times n$ such that $S_{i,j} = 1$ if and only if there exists an edge from vertex v_i and v_j and 0 otherwise. For a weighted graph $S_{i,j}$ is a real number corresponding to the weight $s_{i,j}$ of edge $e_{i,j}$.*

The edge weight $s_{i,j}$ is often treated as a measure of similarity between nodes. Definition 3 and 4 are defined similar to Goyal et al. [13].

Definition 3 (First-Order Proximity)**.** *Edge weights $s_{i,j}$ are also called first-order proximity between nodes v_i and v_j since they are the first and foremost measures of similarity between two nodes.*

Definition 4 (Second-Order Proximity)**.** *The second-order proximity between a pair of nodes describes the proximity of the pair's neighborhood structure. Let $\mathbf{s_i} = [s_{i1}, \dots, s_{in}]$ denote the first-order proximity between v_i and other nodes. Then, second-order proximity between v_i and v_j is determines by the similarity of $\mathbf{s_i}$ and $\mathbf{s_j}$.*

In similar fashion to definition 4, higher order proximities can be defined. Both definition 3 and 4 are so called *pairwise* node proximities, i.e an estimation of similarity between pairs of nodes, no more no less. We define $s_G(v_i, v_j)$ to be a pairwise proximity measure and

$$\mathbf{S}_{i,j} := s_G(v_i, v_j) \tag{3.1}$$

so that the *proximity matrix* $\mathbf{S}$ contains the pairwise node proximities between node v_i and node v_j. A node embedding is when each vertex in a graph or network is mapped to a vector in a vector space. We define a node embedding as [13]:

Definition 5 (Node Embedding)**.** *Given a graph $G = (V, E)$ a node embedding is a mapping $f : v_i \rightarrow \mathbf{z_i} \in \mathbb{R}^d \quad \forall i \in \{1, \dots, n\}$ such that $d << |V|$ and the function f preserves some proximity measure $\mathbf{S}_{i,j} := s_G(v_i, v_j)$ defined on graph G.*

Hence, each node v_i in the graph we have a low dimensional vector $\mathbf{z}_i$. We define

$$\mathbf{Z} \in \mathbb{R}^{d \times |V|} \tag{3.2}$$

to be a matrix where column i corresponds to the embedding $\mathbf{z}_i$ of node v_i.

Attributed Graphs

Some graphs are equipped with attributes. For example, the parameters mentioned in chapter 2.2.3 are attributes of the IT-architecture graph. The definition of an attributed is from [43].

> **Definition 6** (Attributed Graph)**.** *An attributed graph is denoted $G' = (V, E, A)$, composed of a set of vertices V, a set of edges E, and an attribute function $A : V \to 2^{\mathcal{A}}$, where $\mathcal{A}$ is the set of all possible attributes.*

In the IT-architecture graphs described in the simulator (chapter 2.2), typical attributes are which type of asset the vertices are (host, credential, network etc). Other attributes are the values they are connected to, like UNI[1] and AP[2].

3.1.1 Graph Isomorphism

Graph isomorphism is a common issue when dealing with graph comparison. Two graphs are isomorphic if they are equal up to a relabelling of their vertices. In the *Automatic Designer* [22] implementation it is essential that the agent can recognize graph isomorphism. It turns out to be a rather complex problem according to Fortin [11]:

"... it is clearly in NP but is not known to be in P and it is not known to be NP-complete".

Hence, solving the graph isomorphism problem is infeasible, but any choice of embedding should be robust against isomorphic transformations of the graphs.

3.2 Skip-gram

Many graph and node embedding algorithms incorporate the *Skip-gram* model, initially developed for learning embeddings of words, in for example word2vec [33]. The skip-gram is a simple but efficient one layer feed forward neural network. Word embedding is the idea of mapping a word from a sentence to a vector space. The idea is that [36]:
"the words appearing in similar context tend to have similar meanings and hence should have similar vector representations".
The skip-gram model as defined in [34] will be explained here, with details from [31]. The skip-gram is a one hidden layer neural network model that is fully connected. Let $|V|$ be the number of words in the vocabulary, i.e the number of unique words in the dataset. Then, given an input word v, the neural network is trained to predict its neighboring words.

The hidden layer is a weight matrix $W \in \mathbb{R}^{|V| \times d}$, where each row t represents the embedding for the word v_t with dimension d. The input is a one-hot vector to select the corresponding embedded vector in the weight matrix. Given an input word and a context size, the neural network is trained

[1] user needs information
[2] attacker persistence

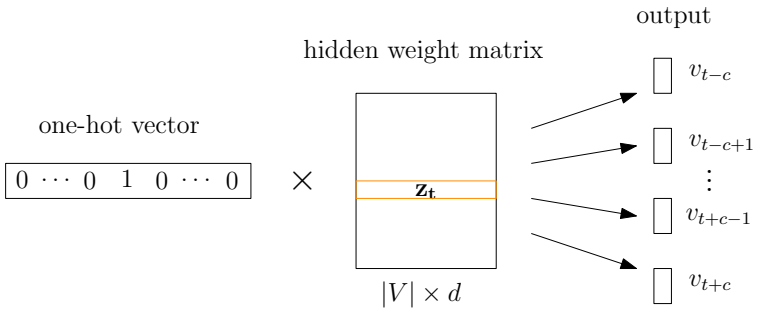

Figure 3.1: *A overview of the skip-gram model. The input word v_t is used to predict its neighboring words, and v_t's corresponding embedded vector is chosen from the weight matrix using a one-hot vector.*

using word pairs in this context. Given a sequence of training words $v_1, v_2, \ldots, v_T$, the goal is to maximize the average log-likelihood:

$$\frac{1}{T}\sum_{t=1}^{T} logP(v_{t-c}, \ldots, v_{t+c}|v_t) \tag{3.3}$$

where c is the size of the training context, and could be a function of the center word v_t. The context word and the target word are assumed to be independent and hence the posterior probability can be calculated as:

$$P(v_{t-c}, \ldots, v_{t+c}|v_t) = \prod_{-c \leq j \leq c, j \neq 0} P(v_{t+j}|v_t).$$

The basic skip-gram defines the probability $P(v_{t+j}|v_t)$ using soft-max function, also called normalized exponential function:

$$P(v_{t+j}|v_t) = \frac{exp(\mathbf{z}_t \cdot \mathbf{z}_{t+j})}{\sum_{i=1}^{|V|} exp(\mathbf{z}_t \cdot \mathbf{z}_i)} \tag{3.4}$$

where $\mathbf{z}_t$ is the vector representation of the word v_t and V is the vocabulary of all the words. In the node embedding setting, the vector representation is the embedding, and the vocabulary is set of all vertices in a graph. When the training is over, the hidden layer represents the embedding of each of the word in the vocabulary, with the desired dimension d. Note that each word has two trained embedding representations, one in the output layer and one in the hidden layer. However, the embedding in the output layer is not used further. The nominator $\sum_{i=1}^{|V|} exp(\mathbf{z}_t \cdot \mathbf{z}_i)$ is a normalizing constant and is typically expensive to compute when the vocabulary grows large, since it has complexity $O(|V|)$ [25]. For instance, a naive approach like logistic regression which in many applications would not be feasible. Two other ways of dealing with the normalizing factor are *Hierarchical soft-max* and *negative sampling*.

3.2.1 Negative sampling

For each word pair sampled in the skip-gram scheme that are trained to ”1”, the equation 3.4 suggests to normalize over all the negative pairs that equals ”0”. The negative sampling method

instead approximates the normalizing factor by using a subset of the negative pairs, i.e *negative samples*. Consequently, for each input pair of words, only a few weights in the hidden weight matrix and output weight matrix are updated.

The probability of a word getting selected as a negative sample is chosen according to *Unigram distribution* [32] - frequent words in the data set are chosen at a higher probability.

3.3 Node Embeddings

A node embedding (as defined in definition 5) is a mapping of each node in a graph to a vector in a vector space. There are several different methodologies for applying node embeddings. The methods described in this chapter are *static*, i.e they do not deal with graphs that evolve over time. Most static node embedding methods can be divided into one of three categories: *factorization* based methods, *random walk* based methods and *deep learning* based methods. In this thesis work, two different node embedding algorithms are implemented. The first one is node2vec [14] by Grover et al. The main argument for this algorithm is its superior performance in the node prediction task in the survey [13]. However, this algorithm does not deal with attributed graphs, and only embeds nodes according to structural similarities. Consequently, a second node algorithm is implemented, namely gat2vec [43] by Skeikh et al. In their paper the gat2vec algorithm performed well on a multilabel node prediction task against various of other embedding techniques. In contrast to the node2vec algorithm, it incorporates node attributes as well as structural similarities.

Most node embedding algorithms that belongs to one of the three categories can be explained by a encoder-decoder framework developed by Hamilton et al. [16], which is introduced in chapter 3.3.1. Both node2vec and gat2vec belong to the random walk category, chapter 3.3.4. However, the other methods will nevertheless be briefly described in chapter 3.3.2 and 3.3.3.

3.3.1 Encoder-Decoder Framework

The encoder-decoder framework developed by Hamilton et al. [16] and is unified way of describing different types of node embeddings. The encoder

$$ENC : V \rightarrow \mathbb{R}^d$$

is a function that maps each node in the graph to a vector $\mathbf{z_i} \in \mathcal{R}^d$ in a low dimensional vector space. The factorization based methods and random walk methods often use a *direct encoding* approach, where the encoder is an "embedding lookup" [16]:

$$ENC(v_i) = \mathbf{Z}\mathbf{v}_i. \tag{3.5}$$

Here $\mathbf{v}_i \in \mathbb{I}_V$ is a vector indicating the column of $\mathbf{Z}$ that corresponds to node v_i. The training parameters for direct encoding is $\Theta_{ENC} = \{\mathbf{Z}\}$. Hence, the embedding matrix is optimized directly. The decoder is a function that maps a set of node embeddings to user-specified graph statistics. Most algorithms use a *pairwise decoder*

$$DEC : \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R}^+ \tag{3.6}$$

that maps a pair of node embeddings to some defined proximity measure[3], that quantifies the proximity of the two nodes in the original graph. This proximity measure could for instance be *First-Order Proximity* in definition 3. Henceforth, with some proximity s_G of the original graph

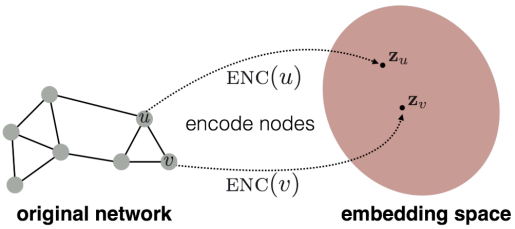

Figure 3.2: *An encoder that maps nodes in a graph to a vector space. Here u and v are nodes in the graph and z_u and z_v their embeddings. Figure taken from [25].*

the aim is to construct the encoder and decoder such that it minimizes the error function:

$$DEC(ENC(v_i), ENC(v_j)) = DEC(\mathbf{z_i}, \mathbf{z_j}) \approx s_G(v_i, v_j).$$

Then the empirical loss function to be minimized is:

$$\mathcal{L} = \sum_{(v_i, v_j) \in \mathcal{D}} \ell(DEC(\mathbf{z_i}, \mathbf{z_j}), s_G(v_i, v_j)) \tag{3.7}$$

where $\mathcal{D}$ is a set of training node pairs and $\ell : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$ is a user-specified loss function. To learn an embedding the following steps needs to be done:

1. Define a proximity measure over the graph G, i.e specify what similarities in the graph that should be preserved in an embedding
2. Define an encoder, i.e a mapping from nodes to embeddings
3. Define an decoder, that reconstructs the proximity values from the embeddings
4. Optimize parameters to minimize the loss function.

3.3.2 Factorization Based Methods

Many factorization based methods employs an encoding algorithm through optimizing a loss similar to 3.7. A factorization based method uses a matrix that in some way represent the connections

[3] a measure of how close the nodes are in G

between nodes, and applies some factorization to it to obtain an embedding. These methods are closely related to dimensionality reduction techniques, eg. eigenvalue decomposition. The factorization based methods often use the *direct encoding* approach, equation 3.5 [16]. One of the earliest methods was the Laplacian eigenmaps (LE) [2], that uses the squared 2-norm as the decoder:

$$DEC(\mathbf{z}_i, \mathbf{z}_j) = ||\mathbf{z}_i - \mathbf{z}_j||_2^2.$$

Another example is the inner product decoder (used in eg. [38] and [5]):

$$DEC(\mathbf{z}_i, \mathbf{z}_j) = \mathbf{z}_i^T \mathbf{z}_j.$$

with a mean squared error (MSE) loss function:

$$\mathcal{L} = \sum_{(v_i, v_j) \in \mathcal{D}} ||DEC(\mathbf{z}_i, \mathbf{z}_j) - s_G(v_i, v_j)||_2^2.$$

The primary difference between these methods is the graph proximity measure $s_G(v_i, v_j)$. It could for example be the the adjacency matrix $s_G(v_i, v_j) := S_{i,j}$ in definition 2, or it could be a power of it $s_G(v_i, v_j) := S_{i,j}^p$ (eg. [5]). These methods are referred to as *Matrix-factorization* methods because of the loss function. When averaging over all nodes they roughly optimize a loss function of the form $\mathcal{L} \approx ||\mathbf{Z}^T\mathbf{Z} - S||_2^2$ where S is a matrix that contains the pairwise proximity measure, which leads to a matrix optimization factorization problem.

3.3.3 Deep Learning Based Methods

In the classical approaches of *factorization* methods and *random walk* methods there are no parameters shared between nodes in the encoder, since they use the direct encoder in equation 3.5. Hence, it can be computationally inefficient since the number of parameters grows $O(|V|)$. The algorithms Deep Neural Graph Representations (DNGR) [6] and Structural Deep Network Embeddings (SDNE) [47] instead incorporate the graph structure into the encoder algorithm. They use autoencoders to compress information about a node's local neighborhood. Instead of using a pairwise decoding approach in equation 3.6, they use unitary decoding:

$$DEC : \mathbb{R}^d \to \mathbb{R}^+.$$

They associate each node v_i with a neighborhood vector $\mathbf{s}_i \in \mathbb{R}^{|V|}$, which corresponds to v_i's row in the proximity matrix $\mathbf{S}$ (defined in eq. 3.1). Furhtermore, they get the following error function [16]:

$$DEC(ENC(\mathbf{s}_i)) = DEC(\mathbf{z}_i) \approx \mathbf{s}_i$$

and loss function:

$$\mathcal{L} = \sum_{v_i \in V} ||DEC(\mathbf{z}_i) - \mathbf{s}_i||_2^2.$$

The encoder and decoder for these approaches consist of multiple stacked neural network layers. Each layer of the encoder reduces the dimensionality of its input and each layer of the decoder increases it. The deep learning methods mostly differ from each other in how they construct the proximity matrix $\mathbf{S}$ and how they optimize the autoencoder[4]. For example, the SDNE [47] looks at the first and second order proximities, definition 3 and 4.

3.3.4 Random Walk Based Methods

The random walk methods use random walk statistics to optimize so that nodes that co-occur over short random walks over the graph have similar node embeddings. The similarity measure is then the probability that nodes v_i and v_j co-occur during a random walk over the network. These methods applies *direct encoding*, equation 3.5. For embedding optimization the following three steps are done [25]:

1. Using some random walk strategy R, do a random walk from each node of the graph.
2. For each node v_i, collect $N_R(v_i)$, the multiset of nodes visited on during the random walk. This set can have repeated elements.
3. Optimize embeddings to maximize likelihood of random walk co-occurence.

When a random walk is done from node v_i and a multiset $N_R(v_i)$ is collected, the random walk method commonly use the skip-gram scheme [14, 43] as described in chapter 3.2. They make the v_i node pose as the center word, and the neighborhood set $N_R(v_i)$ is interpreted as its context. Thereafter, samples from the context is drawn, and they look at the probability of visiting that sample given the center word v_i. The loss function to be minimized, is the negative log likelihood from equation 3.3 [25],

$$\mathcal{L} = \sum_{v_i \in \mathcal{V}} \sum_{v_j \in N_R(v_i)} -log(P(v_j|\mathbf{z}_i))$$

Like in skip-gram scheme, $P(v_j|\mathbf{z}_i)$ is parameterized using soft-max, which leads to:

$$\mathcal{L} = \sum_{v_i \in \mathcal{V}} \sum_{v_j \in N_R(v_i)} -log\left(\frac{exp\{\mathbf{z}_i^T \mathbf{z}_j\}}{\sum_{v_j \in \mathcal{V}} exp\{\mathbf{z}_i^T \mathbf{z}_j\}}\right).$$

The intent is to find embeddings $\mathbf{z}_i$ that minimizes $\mathcal{L}$. The normalizing constant $\sum_{v_j \in \mathcal{V}} exp\{\mathbf{z}_i^T \mathbf{z}_j\}$ is computed in different ways, e.g Node2vec [14] approximates it using negative sampling[5] and DeepWalk [41] hierarchical soft-max.

There are different ways of how to define the random walk. One solution is a fixed-length and unbiased random walk (Deepwalk 2013 [41]), and another is a flexible, *biased*, random walk that can trade off between global and local views of the network (node2vec 2016 [14]).

[4] For further information about autoencoding the reader is referred to Hinton et al. [18], *Reducing the Dimensionality of Data with Neural Networks.*

[5] As described in chapter 3.2.1

3.3.5 node2vec

The node2vec algorithm is the first method that is implemented in this thesis. The node2vec algorithm by Grover et al. [14] is an algorithm that belongs to the *random walk* category. The contribution of node2vec is its flexible and biased random walk strategy R from each node in the graph. The aim is to combine two extreme searching strategies:

- Breadth-first sampling (BFS)
- Depth-first sampling (DFS).

In BFS the random walk is restricted to only nodes in the first order neighborhood, which is closely connected to embeddings that capture structural equivalence. In DFS the random walk is made sequentially with increasing distance away from the source node. As a result, DFS explores a larger part of the graph. The figure 3.3 shows a small example of the BFS and DFS strategies. The random walk described in [14] is as follows. Given a starting node v_s, a random walk of fixed

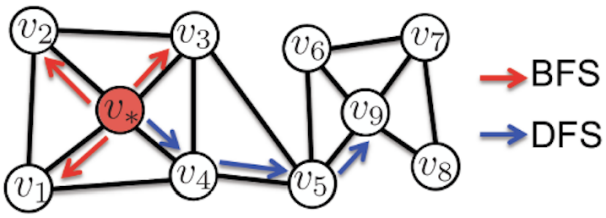

Figure 3.3: *Figure related to the node2vec algorithm. Starting from source node v_*, it is showing a random walk with fixed walk length of $l = 3$ based on the breadth-first search (BFS) and the depth-first search (DFS) strategies [14].*

length l is made. Subsequently, v_j denotes the jth node in the walk starting from $v_0 = v_s$. Then the nodes v_j are generated according to the following distribution:

$$P(v_j = v | v_{j-1} = v_*) = \begin{cases} \frac{\pi_{v_* v}}{Z} & \text{if } (v_*, v) \in E \\ 0 & \text{otherwise} \end{cases}$$

where Z is the normalizing constant approximated with negative sampling (chapter 3.2.1) and $\pi_{v_* v}$ is the unnormalized transition probability between nodes v_* and v that lies on the edge (v_*, v). $\pi_{v_* v}$ is the parameter that decides the random walk strategy. In node2vec, a $2^n d$ order random walk is defined with two parameters p and q. Moreover, consider one step of a random walk starting from node v_s that is now in node v_*. This random walk has to decide which next node to transition to, by evaluating the transition probabilities to the connected nodes from v_*. Then, by letting $\pi_{v_* v_i} = \alpha_{pq}(v_*, v_i)$ the the transition probability is defined as:

$$\pi_{v_* v_i} = \alpha_{pq}(v_*, v_i) = \begin{cases} \frac{1}{p} & \text{if } d_{v_s v_i} = 0 \\ 1 & \text{if } d_{v_s v_i} = 1 \\ \frac{1}{q} & \text{if } d_{v_s v_i} = 2 \end{cases}$$

where $d_{v_s v_i}$ denotes the shortest path distance from starting node v_s to node v_i. For example, in figure 3.4, the distance between v_s and v_1 is $d_{v_s,v_1} = 1$ and as a result $\alpha_{pq}(v_*, v_1) = 1$. Furthermore, α is also referred to as *search bias*. Moreover, the parameters p and q allows the algorithm to combine the BFS and the DFS random walk methods.

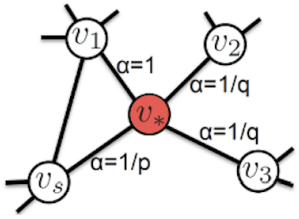

Figure 3.4: *The transition probabilities from node v_* during the random walk, illustrating the search biases α. Here it just transitioned from node v_s to v_*. The return parameter p sets the likelihood of revisiting v_s in the next step and in-out parameter q how likely it is to visit nodes far away from v_s.*

Parameters in node2vec

p: The return parameter p decides the probability of returning to the node just transitioned from. A high value makes it less likely to come back to a node the random walk has just been to.

q: The in-out parameter q decides the probability to visit nodes far away from node v_s. For large q the random walk is biased towards towards nodes that are closer to the starting node v_s.

d: The dimension of the embedded vectors $\mathbf{z}_i$.

γ: The number of walks made from each node.

λ: Walk length per random walk.

k: The maximum amount of nodes the neighborhood set $N_R(v_i)$ can include [14],

3.3.6 gat2vec

The gat2vec algorithm by Sheik et al. [43] is similarly to node2vec in the *random walk* method category, and the second algorithm implemented in this thesis. However, in contrast to node2vec it deals with attributed graphs. Each graph has a structural content, with structural vertices as defined in definition 1. An addition, there exists *content vertices* that are are associated with attributes. The content vertices then have edges to the *attribute* vertices that they are related to. Altogether, given an attributed graph $G' = (V, E, A)$ as in definition 6, the gat2vec algorithm obtains two graphs:

- a connected structural graph $G_s = (V_s, E)$ where $V_s \subseteq V$ is the subset of structural vertices, and every vertex has as least one connection in the set E of edges.

- a bipartite graph $G_a = (V_a, \mathbb{A}, E_a)$ that consists of the vertices $V_a \subseteq V$ associated with attributes and a set of possible attribute vertices $\mathbb{A}$. Lastly, it has a set of edges E_a connecting the V_a and $\mathbb{A}$ that are associated by the function A from definition 6:

$$V_a = \{v : A(v) \neq \emptyset\}$$
$$E_a = \{(v, a) : a \in A(v)\}$$

Figure 3.5 shows an attributed graph where vertex 9 lacks connections in the set E of edges. However, in the corresponding bipartite graph it is second neighbor to vertices 4 and 6. Given

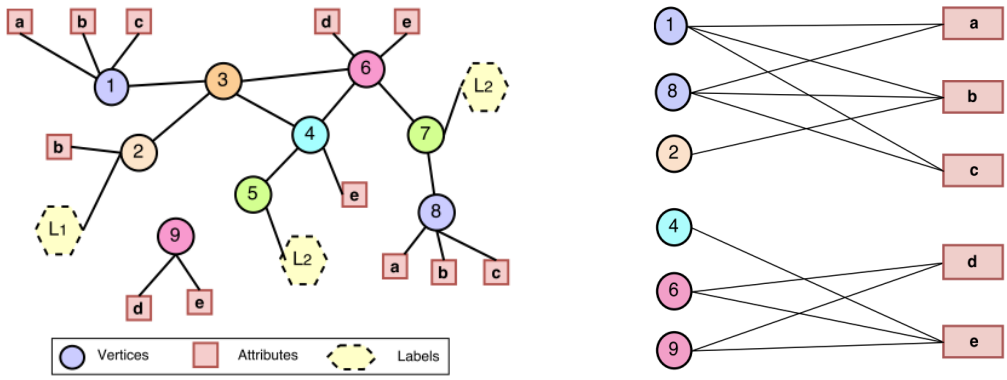

Figure 3.5: *The left side shows an attributed graph $G' = (V, E, A)$. The vertices are numbered and the attribute vertices are lettered. The graph is also partially labelled, i.e a subset of the nodes has a label attached to it. The right side shows the corresponding bipartite graph G_a. Picture from [43].*

this set up, gat2vec performs random walks on both graph G_s and graph G_a. Short random walks are incorporated, as done by DeepWalk [41]. This random walk strategy is a special case of the biased random walk strategy by node2vec (chapter 3.3.5), with the return parameter $p = 1$ and in-out parameter $q = 1$. When doing a random walk over the bipartite graph in figure 3.5 it could

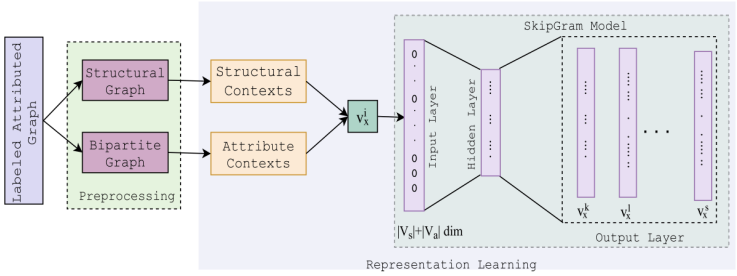

Figure 3.6: *The structure of the gat2vec algorithm. The size of the one hot vector and the hidden layer in the skip-gram model depends on the vertices connected to attributes V_a in addition to the structural vertices V_s. Picture from [43].*

for example be the sequence $[2, b, 1, c, 8, b, 2, b, 8]$. Then, the attribute nodes are skipped, and the kept sequence is $[2, 1, 8, 2, 8]$ with length 5. Thereafter, they random walk sequence of both the attribute graph G_a and the structural graph G_s from each node are incorporated in the skip-gram model. The full architecture of the gat2vec algorithm is shown in figure 3.6.

Parameters for gat2vec

γ: number of walks for each node, on both graphs

λ: the length of the walks

d: Dimension of the embedding

k: Constraint of the neighborhood matrix

3.4 Graph Embedding

There has been a limited amount of research regarding embedding a whole graph to a vector. According to the authors of graph2vec [36] from 2017, they are the first neural embedding approach that learns a representation of a whole graph. In the implementation of the *Automatic Designer* [22] it is of interest to have the option of representing the entire graph as the state in the reinforcement learning setup. It is advantageous to have both the node embedding and graph embedding options available, since it opens up for a larger variety of RL solutions. For this reason, the graph2vec is the third algorithm implemented in this thesis.

3.4.1 graph2vec

Many node embedding rely on the skip-gram model that originates from word2vec [34], that embeds each word in a vocabulary to a vector in a vector space. There is a straight forward extension of this model, called doc2vec [24]. This algorithm does not learn representations of each words, rather it learn embeddings of sequences of words or entire documents. It employs two algorithms, *Paragraf Vector - Distributed Memory* (PV-DM) and *Paragraf Vector - Distributed Bag of Words* (PV-DBOW)[6]. The graph2vec algorithm implements the latter, PV-DBOW, which in principal is similar to the skip-gram model.

In graph2vec, the entire graph is considered a document and they introduce *rooted subgraphs* around each node in the graph to describe the words in the document. In the same manner as node embeddings, structurally similar graphs yields similar embeddings. Henceforth, to get rooted subgraphs $sg_n^{(d_s)}$ for each graph, they follow the Weisfeiler-lehman (WL) [44] relabeling process. The algorithm requires d_s, which is the intended degree of neighbors that will be considered when extracting the subgraph. For each node v and $d_s > 0$, the algorithm starts by collecting its direct adjacency neighbors and letting $d_s += -1$. Then for each of the neighbors, it collects a new set of neighbors until $d_s = 0$. Thereafter, it return a set of node labels for each of the nodes in last set. A more detailed description in algorithm 1, which is as described in the graph2vec paper [36].

[6]Not to be confused with CBOW model - Continous bag of words. Despite similar names, the concept of DBOW has more similarities with the skip-gram model than CBOW.

Given rooted subgraphs for every node in each graph, the setup is as follows. Let $\mathbb{G} = \{G_1, G_2, \ldots, G_k\}$ be a set of graphs, $SG_{vocab} = \{sg_1, sg_2, \ldots\}$ a vocabulary of subgraphs that represent words and $\Phi \in \mathbb{R}^{|\mathbb{G}| \times d}$ a matrix vector representation of graphs. Then, as in the skip-gram scheme the following log likelihood is to be maximized:

$$\sum_{G_i \in \mathbb{G}} \sum_{v \in V_i} \sum_{s=0}^{d_s} logP(sg_v^{(s)} | \Phi(G_i))$$

where $\Phi(G_i)$ is G_i's vector representation. In similar fashion as node2vec, the probability $logP(sg_v^{(s)}|\Phi(G_i))$ is calculated using soft-max and the normalizing constant is dealt with using negative sampling. Furthermore, the updates are made using stochastic gradient descent (SGD).

Algorithm 1: `GetRootedSubgraphs`(v, G, d_s)

Input:
v: The node that is the root of the subgraph
$G = (V, E)$: The graph that v belongs to
d_s: The degree of neighbors considered in the subgraph

Output: A rooted subgraph $sg_v^{(d_s)}$ around node v

$sg_v^{(d_s)} = \{\}$

if $d_s = 0$ **then**
  $sg_v^{(d_s)}$:= the label of node v
  else
    $\mathcal{N}_v := \{v' | (v, v') \in E)\}$
    $\mathcal{M}_v^{(d_s)} := \{$`GetRootedSubgraphs`$(v', G, d_s - 1) | v' \in \mathcal{N}_v\}$
    $sg_v^{(d_s)} := sg_v^{(d_s)} \cup$ `GetRootedSubgraphs`
    $(v, G, d_s - 1) \oplus sort(\mathcal{M}_v^{(d_s)})$
  end
end

return $sg_v^{(d_s)}$

Parameters graph2vec

d: Dimension of the embedding

$\mathbf{d}_s$: Number of WL iterations, i.e the degree of neighboors considered in the subgraphs

α: The learning rate of the SGD

3.5 Summary of the Implemented Algorithms

Three algorithms are evaluated in this thesis work. Two of them are node embeddings, where each node in a graph are mapped into a vector in a d-dimensional vector space. One of these two, node2vec, look at structural similarities and use a biased random walk to capture both 1^{th} and higher order proximities. However, it does not consider the attributes of the graph. The other,

gat2vec, introduces a *bipartite graph* containing the attribute information of the graph. It executes an embedding that regards both the bipartite graph and the structural graph. The third algorithm is a graph to vector embedding, where the entire graph is mapped in to vector. A summary of the algorithms is presented in table 3.1.

Table 3.1: *A summary of properties of the algorithms implemented*

	node2vec	**gat2vec**	**graph2vec**
Embedding type	Node	Node	Graph
Year	2016	2018	2017
Sampling method	Negative sampling	Hierachical softmax with Huffman coding	Negative Sampling
Proximity measure	structural equivalence and $1-k^{th}$ order proximities	$1-k^{th}$ order proximities on G_s and G_a	$1-d_s^{th}$ order proximities
Attributed graphs	No	Yes	Yes, but only labels
Method	Biased Random Walk	Random Walk	WL-kernels

Chapter 4

Methodology

In this chapter the methods for obtaining data, experiments performed and implementation strategies will be described. The task of predicting the TTC values given an embedding is made into a classification problem, which is described in chapter 4.1. Further, to investigate graph and node embedding algorithms on the IT-architecture graphs in the simulator, data is required. In the simulator[1] IT-security graphs can be created manually. How data is obtained is described in chapter 4.2, along with the corresponding pseudo algorithms. The analysis of the data and how it is made to fit into the classification setup is described in chapter 4.2.1. Thereafter, chapter 4.3 briefly describes the statistical methods that are implemented. Furthermore, the classifiers that are used to predict the TTC values are described in chapter 4.4. Lastly, the implementation of node2vec, gat2vec, and graph2vec is presented in 4.5, and chapter 4.6 presents the baseline algorithms that are used for comparison.

4.1 Classification Setup

This thesis uses the simulator, which is a simplified version of the securiCAD. The concept to investigate is of how well the embeddings can capture characteristics of the IT-architecture, although an IT-architecture generated by the simulator. The simulator can calculate TTC values for each node in a graph, as described in chapter 2, which is used to evaluate the performance of the embeddings. The TTC value to each node is divided to one of five classes, namely if it is; *very easy*, *easy*, *hard*, *very hard* or *impossible* for the attacker to reach the node.

The TTC value for each attack step is generated by the Monte Carlo estimate of samples from the gamma distribution equipped to that attack step (see chapter 2.2.2). For the sake of testing the concept, it is necessary that the TTC values do not vary noticeably between two different attack calculations on the very same graph. Rather, it is advantageous if it is close to deterministic. In addition, since the simulator is not reality true and the attack calculations are expensive, it

[1] described in section 2.2

is preferable not having to run multiple attacks for each graph. For this reason, rather than averaging over many samples, one single sample is drawn for each attack step, with a variance set close to zero. Henceforth, in the implementation a single sample is drawn according to the probability distribution in equation 2.1, chapter 2.2.2, with the scale parameter $\theta \to 0$ such that $Var(X) = k\theta^2 \approx 0$. Moreover, the shape parameter k is set such that an legitimate attack step gives $TTC \sim 1$, bypassing firewall is $TTC \sim 50$ and access by exploit gives $TTC \sim 100$.

Legitimate:	$E[X] = k\theta = 1$	$Var(X) = k\theta^2 \approx 0$
Bypass firewall:	$E[X] = k\theta = 50$	$Var(X) = k\theta^2 \approx 0$
Access by exploit:	$E[X] = k\theta = 100$	$Var(X) = k\theta^2 \approx 0$

Table 4.1: *Table of expected TTC values for different attack steps, where the TTC value for an attack step is modelled as a $X \in \Gamma(k, \theta)$ random variable.*

Henceforth, if the attacker can reach a node through legitimate attack steps exclusively, it falls into the category *very easy*. If the attacker has to pass a firewall, it is *easy*. One illegitimate attack step or two firewalls is *hard*, and *very hard* when there is one more illegitimate step. Lastly, the *impossible* category is for all nodes with $TTC > 195$.

Table 4.2: *The thresholds for the total TTC categories.*

Category	very easy	easy	hard	very hard	impossible
Total TTC	<45	<95	<145	<195	≥195

In summary, the prediction task is made into a multiclass classification problem consisting of five classes.

4.2 Producing Training Data

IT-architecture graph can be build in the simulator and to get enough data to conduct experiments, data is generated randomly. Given the maximum number of entities allowed of each type, algorithm 2 was used to set the number of entities in each graph. For each entity type except the user and the attacker, a random number between 0 and 15 is drawn, with some few restriction. For example, there is a limitation that the instances of dataflows, informations, and credentials cannot be more than the number of hosts in one graph, neither can it be less than 2 hosts and 1 network.

Subsequently, the IT-architecture graph was generated according to algorithm 4. Initially, all the networks are created and randomly connected to each other. Thereafter, all hosts are connected to random networks. Subsequently credentials are connected to random hosts, followed by informations and dataflows. Lastly, the attacker and the user are similarly connected to one random host, not necessarily the same. When all the assets are connected, the attack simulations on the graph is made to achieve the true TTC-values for each node in the graph.

The final number of graphs used is 1000, and the maximum number of entities of each type is is set to 15. The training data is generated by iterating algorithm 2, 4 and calculating its TTC-

values.

Algorithm 2: Generating the number of graph entities

Input: Max networks, Max hosts, Max credentials, Max Informations, Max dataflows
Output: Number of entities in an IT architecture graph

```
Networks = Unif{1, Max networks}, Hosts = Unif{2, Max hosts}
if Max Credentials > Hosts then
    Credentials = Unif{0,Hosts}
    else
        Credentials = Unif{0,Max Credentials}
    end
end
if Max information > Hosts then
    Informations = Unif{1,Hosts}
    else
        Informations = Unif{1,Max informations }
    end
end
if Max dataflow > Hosts then
    Dataflows = Unif{0,Hosts}
    else
        Dataflows = Unif{1,Max dataflow}
    end
end
return Networks, Hosts, Credentials, Informations, Dataflows
```

Generate training data by iterating:

1. Decide the number of entities of each type according to algorithm 2
2. Generate a graph $G = (V, E)$ according to algorithm 4
3. Run attack simulations on G and store the TTC value for each node

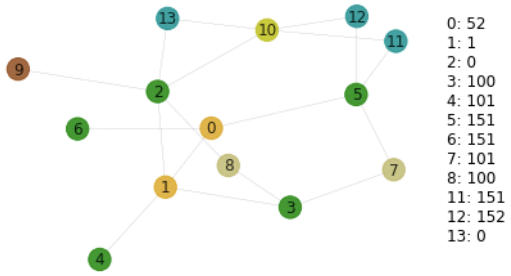

Figure 4.1: *An example of randomly generated graph. The numbers are the node id's followed by its calculated TTC value. The node id's corresponds to the following entities: 0-1=Networks, 2-6=Hosts, 7-8=Credentials, 9=Attacker, 10=User, 11-13=Informations (no dataflows in this particular example). The nodes that do not have a TTC value is either the attacker (id 9) or the user (id 10).*

Figure 4.1 and figure 4.2 present to smaller examples of randomly generated graphs according to the algorithms 2 and 4. The numbers corresponds each node's id, and it is followed by its total TTC value. Two of the nodes in both figures do not have a corresponding TTC value, which are the nodes associated to the user and the attacker.

Algorithm 4: Generating a graph of an IT-architecture in the simulator

Input: Networks, Hosts, Credentials, Informations, Dataflows
Output: Graph of an IT-architecture

$V = \{\emptyset\}$, $E = \{\emptyset\}$ `// An empty vertex and edge set`

for *Networks* **do**
  Create a network vertex v_N
  if $V \neq \emptyset$ **then**
    Draw a vertex v from V
    Add an edge $e = (v_N, v)$ to E
  end
end
for *Hosts* **do**
  Create a host vertex v_H
  Draw a network vertex v_N from V
  Add an edge $e = (v, v_N)$ to E
end
for *Credentials* **do**
  Create a credential vertex v_C
  Draw two host vertices v_{H1},v_{H2} from V
  Add a **store** edge $e = (v_C, v_{H1})$ to E
  Add a **grant access** edge $e = (v_C, v_{H2})$ to E
end
for *Informations* **do**
  Create information vertex v_I
  Draw a host vertex v_H from V
  Add edge $e = (v_I, V_H)$ to E
end
for *Dataflows* **do**
  Create a dataflow vertex v_D
  Draw two host vertices v_{H1},v_{H2} from V
  Add edge $e = (v_D, v_{H1})$ and edge $e = (v_C, v_{H2})$ to E
end
Create an attacker vertex v_A and an user vertex v_U
Draw two hosts v_{H1}, v_{H2} with replacement from V
Add edge $e = (v_A, v_{H1})$ and $e = (v_U, v_{H2})$ to E

return $G = (V, E)$

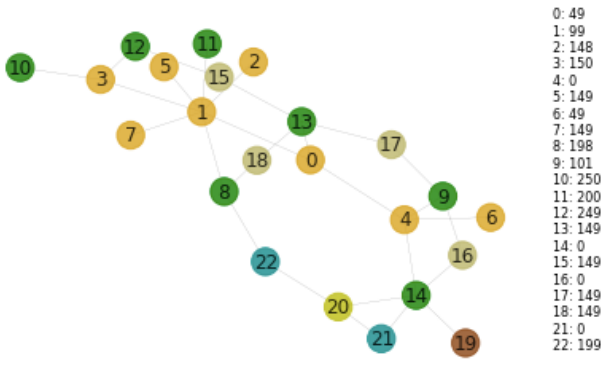

Figure 4.2: *Another example of randomly generated graph. The numbers are the node id's followed by its calculated TTC value. The node id's corresponds to the following entities: 0-7=Networks, 8-14=Hosts, 15-18=Credentials, 19=Attacker, 20=User, 21-22=Informations (no dataflows in this particular example). The nodes that do not have a TTC value is either the attacker (id 19) or the user (id 20).*

4.2.1 Data Analysis

The same 1000 randomly generated graphs are used for the implementation of all three algorithms. The figure 4.3 displays how the TTC values are distributed over all nodes over all graphs. The most common class is *easy*, which has more than 7000 members. The second largest class is *hard*, with almost 7000 members. The right histogram in figure 4.3 shows the percentage of members in each class out of the whole data set.

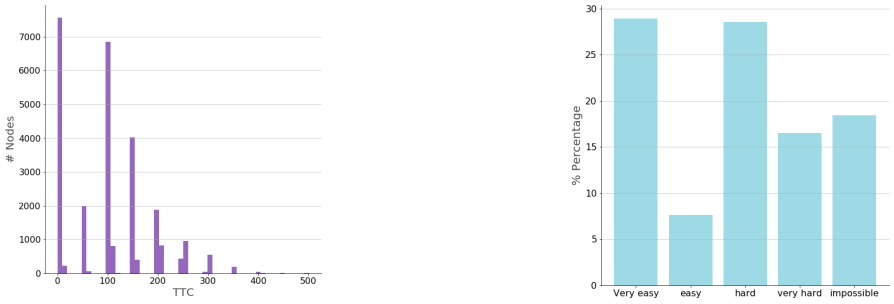

Figure 4.3: *The left figure displays a histogram of the node TTC values with the total number on the y axis. The bar plot shows the corresponding classes, very easy:* $TTC < 45$*, easy:* $45 \leq TTC < 95$*, hard:* $95 \leq TTC < 145$*, very hard:* $145 \leq TTC < 195$*, impossible:* $195 \leq TTC$*.*

For each graph, the average TTC value is likewise calculated. The result is shown in graph 4.4. When averaging over all nodes in a graph, there is a heavier weight on the *hard* class than the others, which has more than 35% of the whole population.

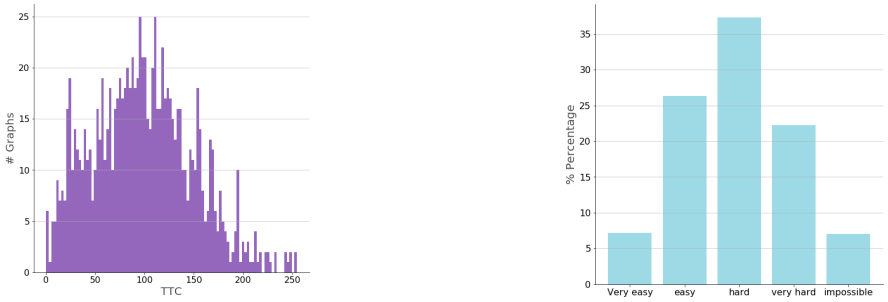

Figure 4.4: *Histogram of the average node TTC value in the graphs. The bar plot shows the corresponding classes, very easy:* $TTC < 45$*, easy:* $45 \leq TTC < 95$*, hard:* $95 \leq TTC < 145$*, very hard:* $145 \leq TTC < 195$*, impossible:* $195 \leq TTC$*.*

The number of members in each class is significantly lower when using the averaging method.

4.3 Statistical Methods

This section briefly describes some statistical concepts that are used to predict the TTC values given the embeddings.

4.3.1 Supervised versus Unsupervised Learning

Most statistical learning tasks can be divided into three categories, the *supervised*, *semi-supervised* and the *unsupervised* [21]. In Supervised learning there exists an outcome variable that can guide the learning process. We have for each observed data point x_i an associated response measurement y_i. Hence, given a data set $\{(x_1, y_1), \ldots, (x_n, y_n)\}$ we want to find an estimate $\hat{f}$ that relates the observation point to the response variable. This could either be to accurately predict the measure y for future observations (prediction), or to gain a better understanding of the relationship between the observation and the response variable (inference).

In unsupervised learning the data does not have a target variable, only features are observed and there is no measurement of the outcome. In these cases the goal is rather to find how the data is organized or clustered [17]. The *semi-supervised* category consists of a mixture of these two, some observed data points x_i has a response measure y_i whereas others do not.

The embedding algorithms that are implemented in this thesis are all in the unsupervised category. Many of the embedding techniques can be made in an semi-supervised manner. This is done when e.g the authors intend to classify the node labels of a graph given the embeddings, and incorporates the information of a few correct node labels into the embedding algorithm. Naturally, this improves the classification result in most cases. However, one of the research topics in this thesis is if the embeddings can capture characteristics of the graph such that the TTC values can be extracted, even if it is done unsupervised.

Contrary to the embedding algorithms, the classification part is completely supervised. The classification is done by using the embedding of a node as the data point x_i, and the class it belongs to as its response measure. Likewise with the graph2vec algorithm, the data point is the graph embedding and its response measure is the class that graph belongs to.

4.3.2 The Classification Setting

In chapter 4.1 there is an explanation of how the TTC values are divided into classes of *very easy*, *easy*, *hard*, *very hard* and *impossible*. In the classification setting, the aim is to fit a model with as small error as possible. One common way to quantify the accuracy of of the estimate $\hat{f}$ is the training *error rate*

$$\frac{1}{n}\sum_{i=1}^{n}\mathbb{I}(y_i \neq \hat{y}_i)$$

where $\mathbb{I}$ is the indicator function, $\hat{y}_i$ is the predicted class of an observation, y_i is its true class, and n the total number of observation. Hence, $\mathbb{I}(y_i \neq \hat{y}_i)) = 1$ each time a false prediction is made, and thus the error rate should be minimized.

4.3.3 Testing and Training

In many circumstances it is not good enough to calculate the error rate of $\hat{f}$ on the same data that was used for training. To have a measure on how good the model actually is the accuracy is deduced on a new test data set that was not used for training [21]. To get access to this it is common to divide the data into a test set and a train set, where the train set is used for training the model and the test set for testing its accuracy.

In this work 10% of the data is used for testing, i.e 100 out of the 1000 graphs in the generated data. The same division of test set and train set is made for all three of the algorithms. It is common to use cross validation[2] for dividing the data into test and train set and report error averages. This is done to make sure that there is no particular bias in the test set. However, since all the graphs in the data are randomly generated in the same fashion, the same test set is used for all trials in all three algorithms.

4.3.4 Cross Validaton for Parameter Tuning

Cross validation is to divide the data set into smaller regions, and train a model using all but one of these sections. This is done for each section, and the prediction error result is averaged among them. In K-fold cross validation, the data set is divided into K sections. Each model has some parameters that need to be optimized for the particular dataset at hand. Henceforth, the cross validation procedure can be executed for a range of values of the model parameter. This achieves a prediction error curve as a function of the tuning parameter, and the optimal parameter choice.

The function GridSearchCV [40] implements this method from the scikit-learn package, and is used for parameter tuning in the classification models in this thesis.

4.3.5 Classification Score

There are different measures of evaluating classification tasks. The table below is a performance table, which shows the possible outcomes when trying to predict whether a set of observations belongs to a class or not.

		Predicted **Negative**	Predicted **Positive**
True	**Negative**	True Negative (tn)	False Positive (fp)
	Positive	False Negative (fn)	True Positive (tp)

In the binary classification setting, it shows the amount of correct predictions, true negatives and true positive, and false predictions, false negative and false positive. The predicted accuracy is

[2] Described in chapter 4.3.4

computed as:

$$A = \frac{tp + tn}{tp + fp + fn + tn} = \frac{tp + tn}{N}$$

where N is the total number of observations in the dataset. The accuracy is the fraction of correctly classified observation, both positives and negatives. In addition, two other common measures called *precision* and *recall* can likewise be computed:

$$P = \frac{tp}{tp + fp}, \qquad R = \frac{tp}{tp + fn}.$$

The precision is the fraction of correctly classified observations among all the observations that were predicted as positives. Moreover, the recall is the fraction of correcly classified positives among all true positive observations. This is in a binary setting, i.e when there is only two classes. In the multiclass setting these measures are averaged among all the classes. For the precision and the recall there are two common averaging methods, the *micro* and the *macro*. For K number of classes, the *micro average* is

$$P_{Micro} = \frac{\sum_{k=1}^{K} tp_k}{\sum_{k=1}^{K} (tp_k + fp_k)}, \qquad R_{Micro} = \frac{\sum_{k=1}^{K} tp_k}{\sum_{k=1}^{K} (tp_k + fn_k)} = \frac{\sum_{k=1}^{K} tp_k}{N}.$$

Here R_{Micro} is in fact the same as the accuracy in the multiclass setting [46]. The *macro average* is:

$$P_{Macro} = \frac{\sum_{k=1}^{K} P_k}{K}, \qquad R_{Macro} = \frac{\sum_{k=1}^{K} R_k}{K}$$

The macro averaging gives equal weight to each class, whereas large classes dominate the small classes in micro averaging. In this thesis, both methods are implemented for comparison. A third less common averaging method alters the *macro* method by accounting for label imbalance:

$$P_{weighted} = \frac{\sum_{k=1}^{K} n_k P_k}{N}, \qquad R_{weighted} = \frac{\sum_{k=1}^{K} n_k R_k}{N}$$

where n_k is the number of predicted instances of class k. The weighted average does not necessarily fall between the *micro* and the *macro* methods, and is also implemented. The F_1-score used in this thesis is the harmonic mean of the averaged precision and recall:

$$F_1 = 2 \cdot \frac{P_{average} \cdot R_{average}}{P_{average} + R_{average}}$$

where *average* is the micro, macro or weighted average.

4.4 Classifiers

In this thesis two classifiers are implemented to predict the TTC values. On the node classification task (node2vec and gat2vec), the decision tree classifier is used. In the graph classification task (graph2vec), the support vector classifier is instead applied.

4.4.1 Support Vector Classifier (SVC)

A support vector machine (SVM) is classifier that can deal with non linear boundaries between classes. For example, it could use a kernel that is polynomial or radial to divide the feature space [21]. However, in the implementation of graph2vec [36] the authors employ the SVM with a linear kernel. This particular case of SVM is called Support Vector Classifier (SVC), and to follow their methodology this is also what is chosen in this work. In the binary class setting, a SVC draws a hyperplane in the feature space to separate the classes. It is sometimes called a *soft margin classifier* since it allows for some miss-classification in the data [21]. In a d -dimensional setting, a hyperplane is defined as

$$f(X) = (\beta_0 + \beta_1 X_1 + \beta_2 X_2 + \dots \beta_d X_d) = 0.$$

In a data set $\{x_i, y_i\}_{i=1}^{N}$, with $y \in \{TRUE, FALSE\}$, all observations that fulfills $f(x) > 0$ will be classified to belong to $y = TRUE$. Similarly, the ones that fulfills $f(x) < 0$ will be classified to the other class $y = FALSE$. Moreover, the distance for an observation to the hyperplane can be interpreted as the confidence that the observation is correctly classified.

When constructing a support vector classifier, it is not certain whether or not the training data set is linearly separable. Therefore, it is allowed to missclassify some data points. For this purpose a tuning parameter C is introduced. The tuning parameter decides to what extent missclassification is allowed. Henceforth, this parameter is set using the grid search cross validation as described in chapter 4.3.4.

The SVC is limited to binary classification, and is not naturally extended to multiclass classification tasks. One solution is the *One-Versus-the-Rest* classification, which is the one implemented in the Scikit-learn [40] package. In this solution, one SVC is fit for each of the K classes, where a hyperplane $f_k(X)$ is created between class k and all the other classes. A test observation x is assigned to the class for which $f_k(x)$ is the largest. Henceforth, this value amounts to the certainty that the observation x belongs to class k rather than any of the other classes [21].

4.4.2 Decision Tree Classifier

The Decision Tree Classifier (DTC) is a tree-based method that partitions the feature space into regions. Subsequently, it fits a simple model, like a constant or a class, into each region [17]. Figure 4.5 shows a small example of a decision tree with two features, X_1 and X_2. Here the splits $t_1, \dots, t_4$ are made giving the regions $R_1, \dots, R_5$. The new data points that falls into region R_m are classified to the majority class of the training data in that region. For example, if in the training data only has nodes of type *easy* in region R_1, each new observation with an embedding that falls into that region will likewise be classified to *easy*.

There are different ways to grow the tree, i.e to decide where to make the next split. In this thesis the *Gini index* is applied as a measure of quality of a specific split. The Gini index is defined

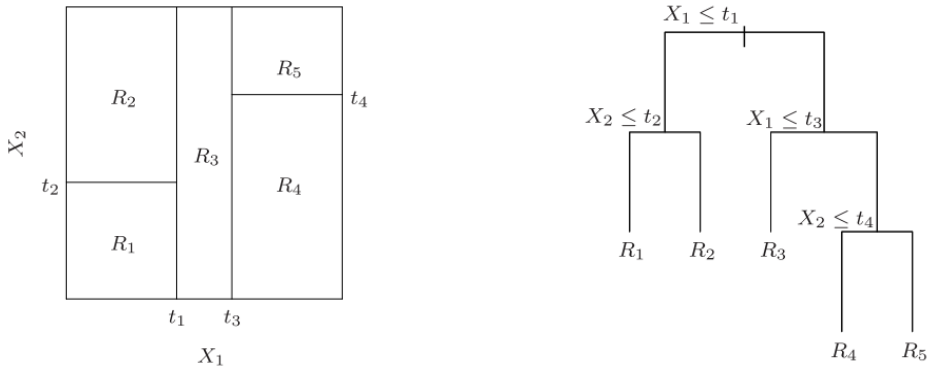

Figure 4.5: *This is an example of tree splitting with a two dimensional feature space. Here t_1 is the first split, t_2 the second and so forth, and we end up with 5 regions, $R_1, \ldots, R_5$.*

as:

$$G = \sum_{k=1}^{K} \hat{p}_{mk}(1 - \hat{p}_{mk}) \tag{4.1}$$

where K is the number of classes in the data set and $\hat{p}_{mk}$ is the proportion of training observations in the mth region from the kth class, such that

$$\hat{p}_{mk} = \frac{\text{number of observations from class } k \text{ in region } m}{\text{total number of observations in region } m}.$$

The Gini index is close to zero when either $\hat{p}_{mk} \to 0$ or $\hat{p}_{mk} \to 1$ for all k, therefore it is also referred to as a measure of node purity [21]. As a result, when the Gini index is low it indicates that most observations in each region belongs to the same class and thus they are pure.

The splitting will continue until either all the regions are completely pure, or when a certain stopping criteria is met. In this thesis the stopping criteria *minimum samples split* is used, which is the minimum amount of members in a region to allow for a split. The reason to have this constraint is to prevent overfitting. Overfitting is the concept of having a model that is too fitted to the training data but does not generalize well to unseen data, and it is a common phenomenon in statistical models.

Yet anther way to control for overfitting and improve the predictive accuracy is to use random forests [4], which is likewise implemented in this work. A random forest grows an ensemble of decision trees on various sub-samples of the dataset, and lets them vote for the most popular class for each new observation. With a big enough ensemble, random forests do not overfit the data [4]. When building the decision trees in a random forest, only a random subset of the features are considered in each split. The amount is typically around $\sqrt{d}$, where d is the total numbers of features (or in this case the dimension of the embedding that is being evaluated). This process is to decorrelate the trees and reduce variance [21].

The implementation of the random forest classifier is done using the sklearn package [40], and the minimal samples split paramenter is tuned using the grid search cross validation described in chapter 4.3.4.

4.5 Implementation of Algorithms

All implementation is done in the python 3 language. Each algorithm has different hyper parameters, and the tuning of them is done by evaluating classification scores when classifying the TTC values in the test data. Initially, a graph or node embedding is done on the dataset with 1000 graphs with some arbitrary parameter setting. Thereafter, the TTC values for each node in each graph is computed. Subsequently, the TTC values are predicted using the DTC as described in chapter 4.4.2 or the SVC, chapter 4.4.1. Finally, the F1 score is evaluated for different hyper-parameters, and they are updated accordingly. Clearly, it is optimal to implement a grid search over all parameters at the same time, i.e try all possible combinations. However, due to expensive computations one parameter or two parameters are updated at a time.

The node embedding algorithms incorporate random walks, which means the resulting embeddings are slightly different each time on the same data set. For this reason, after their hyper-parameters are set, they are executed 10 times and the average result is reported in the results chapter 5.

4.5.1 node2vec

For the implementation of node2vec a python package GEM [12] is used with modifications, which is built on the software library by [27]. In chapter 3.3.5, there is a description of the model parameters to be set, $(p, q, d, \gamma, \lambda, k)$. The classification method used is the DTC, with the random forest implementation and a grid search cross validation over the minimal split parameter, where $MSS \in \{2, 20, 40, 60, 80, 100, 120\}$. Figure 4.6 displays some tests to decide the the dimension d and neighborhood set k. The grid search is over $d \in \{4, 8, 16, 32, 64, 128\}$ and $k \in \{2, 5, 10, 15\}$, and these parameters was tested separately. The result suggests to use dimension $d = 64$ and neighborhood size $k = 10$.

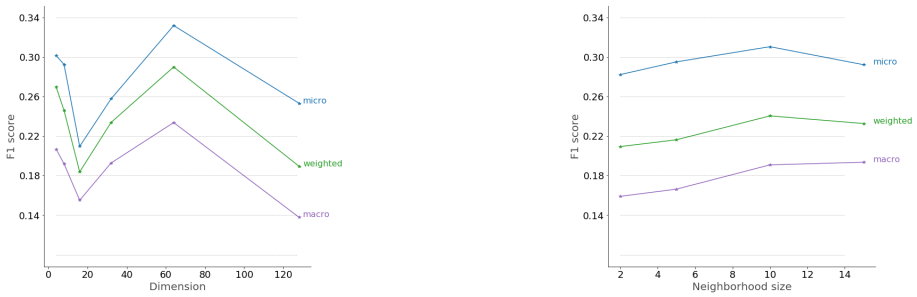

Figure 4.6: *The left figure displays a F1 scores for deciding dimension* d *of the node2vec algorithm. Here the dimension 64 appears to be superior. The right to decide the size of the neighborhood set* k*, where size 10 is slightly better.*

To follow the methodology of Grover et al. [14], the grid search for the parameters q and p is over $p, q \in \{0.25, 0.5, 1, 2, 4\}$. and the result is displayed in figure 4.7. In the F_1-MICRO and F_1-WEIGHTED scores, the choice of letting $(p, q) = (2, 2)$ is superior, whereas the F_1-MACRO suggests to use $(p, q) = (4, 1)$. The parameters are set to $(p, q) = (2, 2)$ since that combination is a good

contender even in the F_1- MACRO scoring.

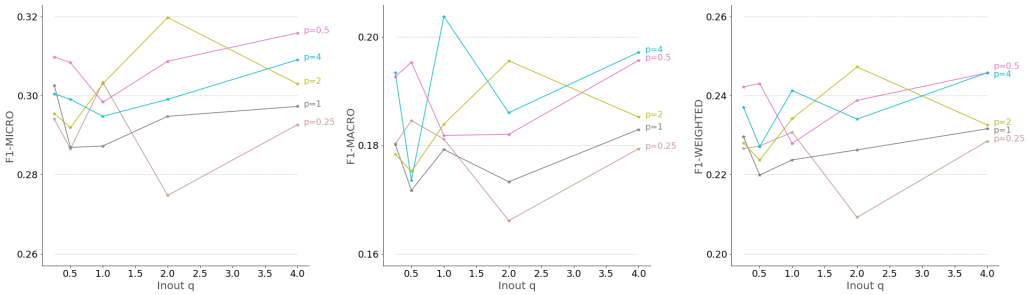

Figure 4.7: *Some test runs for deciding the return-parameter p and the inout-parameter q for the node2vec algorithm. The micro and weighted averaging show that $(p, q) = (2, 2)$ is a good choice. In the macro averaging those parameters are likewise good contenders, even if $(p, q) = (4, 1)$ performs better.*

To decide the walk length λ and number of walks γ, they are tried against each other over grids of $\lambda \in \{20, 40, 60, 100, 120\}$ and $\gamma \in \{5, 10, 15, 20, 25\}$. The figure 4.8 displays the result. In all three averaging methods, the choice of $(\lambda, \gamma) = (60, 10)$ is superior.

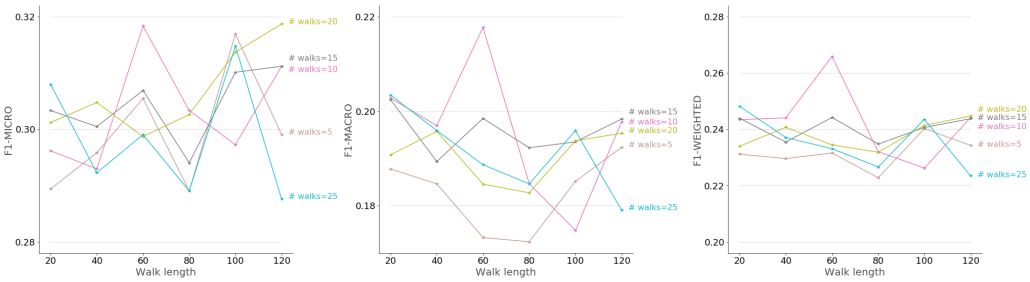

Figure 4.8: *Some of the tests to decide the walk length λ and the number of walks γ for the node2vec algorithm. In all three averaging methods, the choice $(\lambda, \gamma) = (60, 10)$ gives the highest F_1 score.*

In summary, the final parameter setting for the node2vec algorithm can be seen in the table below.

	p	q	d	γ	λ	k
node2vec	2	2	64	10	60	10

4.5.2 gat2vec

The implementation of gat2vec is built on the repository GAT2VEC [42]. To implement the gat2vec algorithm as described in [43], a bipartite graph G_a is required containing all the attributes for all the entities in the simulator. In the gat2vec paper, the labels of each node is partly known and they investigate performing label classification prediction using semi-supervised learning, i.e incorporating the partly known labels when training the embedding. The corresponding labels in this work is the node entity type, for example if a node is a *host* or a *network*. However, in this implementation the labels are treated as attributes of the graph, and thus a part of the bipartite graph G_s. However, the TTC values of each node are *not* used as an attribute when training the

embedding. All the attributes used are presented in table 4.3. The table displays which type of entities that can be connected to the specific attribute, since all entities are not allowed to have all the attributes. Naturally, each entity is connected to its corresponding node type attribute, e.g hosts are connected to the attribute Host. Furthermore, each unique attribute is assigned an attribute id. Some attributes have values, in which case each value of the attribute is assigned an unique id. For example, the attribute *UNI* has values 500, 1000, 2000. Therefore, $UNI500$, $UNI1000$ and $UNI2000$ are all assigned unique attribute id's.

Table 4.3: *A table of the attributes implemented in the gat2vec algorithm, with the corresponding connected entity types. Some attributes have values, in which case each value is treated as an unique attribute.*

Attribute	**Connected entity types**	**Values**
Host	hosts	-
Network	networks	-
Dataflow	dataflows	-
Credential	credentials	-
Information	informations	-
Attacker	attacker	-
User	user	-
User needs information (UNI)	user, informations	500, 1000, 2000
Cost of Compromise (COC)	informations	500, 1000, 2000
Attacker Persistence (AP)	attacker	500, 1000, 2000
Grant Access	hosts, credentials	-
Store	hosts, credentials	-

Further, the COC, AP, Grant Access, and Store are likewise implemented as attributes of the graph. The credentials have a *store* edge to one host, and a *grant access* edge to another. Instead of treating these as special edges between the entities, they are connected using a normal *connect* edge. Instead, the store and grant access are incorporated as attributes, and connected to the hosts and credentials that had these connections between themselves.

The classification method used for gat2vec is DTC implemented with random forest and grid search cross validation over the minimal sample split parameter, where $MSS \in \{2, 20, 40, 60, 80, 100, 120\}$. The parameters to be set are (γ, λ, d, k), with a further description in chapter 3.3.6.

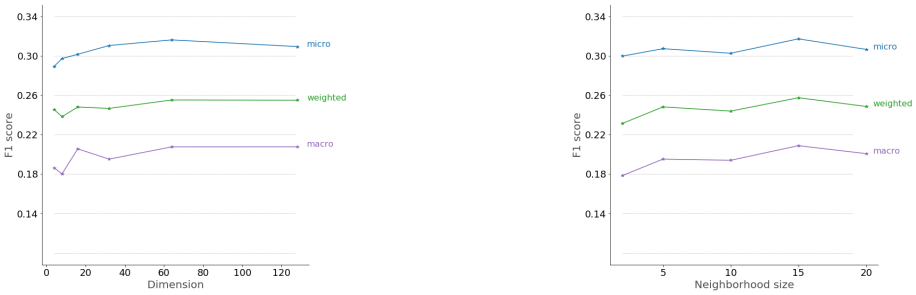

Figure 4.9: *The left figure shows the F_1 scores for the three different averaging methods, as a function of the embedding dimension d. The rightmost figure does similarly for the size of the neighborhood set k. These results suggests to use $d = 64$ and $k = 15$.*

The dimension d is tested over $d \in \{2, 8, 16, 32, 64, 128\}$, and figure 4.9 suggest that the optimal

dimension size is 64. Further, the size of the neighborhood set k is tested over $k \in \{2, 5, 10, 15, 20\}$ and the left figure in 4.9 displays a superior result for $k = 15$. Subsequently, the parameters λ and γ are tested in a combined grid search, where $\lambda \in \{10, 20, 40, 60, 100, 120\}$ and $\gamma \in \{5, 10, 15, 20, 25\}$. The result is shown in figure 4.10, and the figure suggests a general negative correlation of the walk length and the F_1. Thus, the walk length λ is set to 10, and the number of walks γ is set to 20 since the combination $(\lambda, \gamma) = (10, 20)$ performs well in each of the averaging methods.

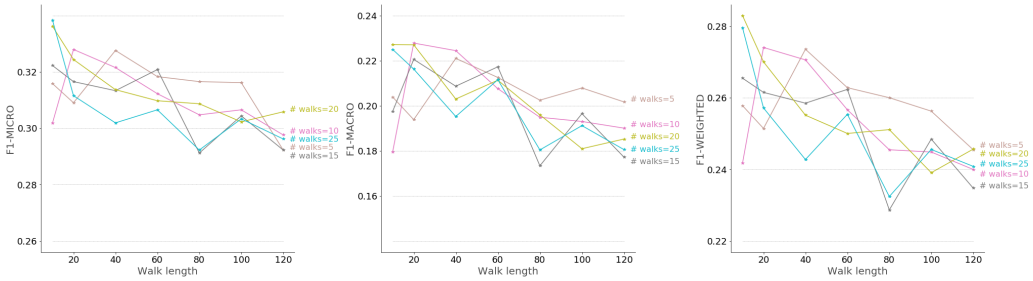

Figure 4.10: *The left, middle and right figure present the F_1-MICRO, F_1-MACRO and F_1-WEIGHTED for a grid search combining the walk length λ and the number of walks γ for the gat2vec algorithm.*

The final parameter setting is the in the following table.

	d	k	λ	γ
gat2vec	64	15	10	20

4.5.3 graph2vec

The graph2vec algorithm is implemented using the repository `graph2vec_tf`, which is a tensorflow implementation by the authors of the paper of graph2vec [36]. The classification method used is the SVC, with a grid search cross validation over the penalty parameter C, letting $C \in \{0.01, 0.1, 1, 10, 100, 1000\}$. The parameters to be set for the graph2vec are (d, d_s, α), which are described in chapter 3.4.1. Figure 4.11 presents the F_1 score using all three averaging methods as a function of the learning rate α, with $\alpha \in \{0.025, 0.1, 0.3, 0.5, 0.75\}$. From this result, the learning rate is chosen to $\alpha = 0.3$.

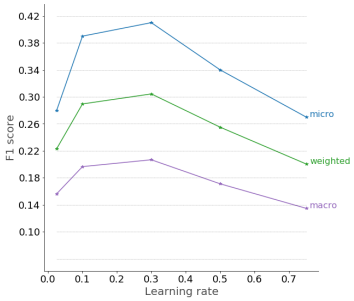

Figure 4.11: *The F_1-MICRO, F_1-MACRO, and F_1-WEIGHTED as a function of the learning rate α. The result displays a maximum for $\alpha = 0.3$.*

The parameters d and d_s are chosen with a combined grid search, letting $d \in \{32, 64, 128, 256, 512, 1024\}$ and $d_s \in \{1, 2, 3, 4, 5\}$. The resulting F_1 score is presented in figure 4.12, with one plot for each of the three averaging methods.

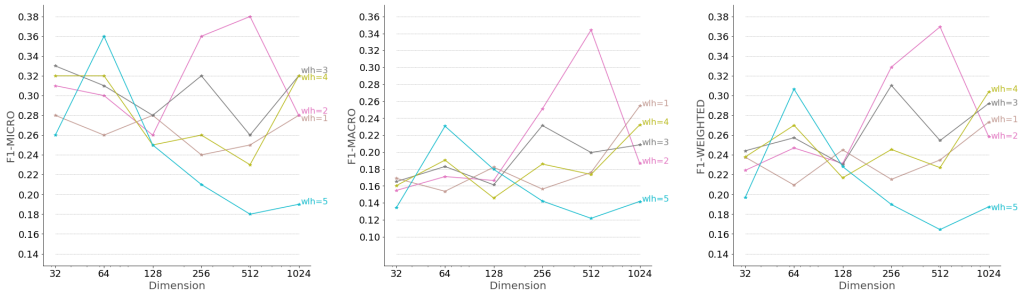

Figure 4.12: *The F_1-MICRO, F_1-MACRO, and F_1-WEIGHTED with varying dimension d and number of WL-iterations d_s. The parameter combination $(d, d_s) = (512, 2)$ performs superior than any other choice in all three averaging methods.*

The final parameter setting for the graph2vec algorithm is presented below.

	d	d_s	α
graph2vec	512	2	0.3

4.6 Baseline Algorithms

Since there are no previous results of this specific task, two naive approaches are implemented for comparison. The two approaches are a *random baseline* and a *constant baseline*.

Random baseline This algorithm guesses randomly among all classes, with uniform distribution.

Constant baseline This algorithm guesses on the same class for every node. For evaluation purposes, it is forced to guess on the class that has *the most* amount of members in it among the test data. As a result, with 5 classes, the total accuracy of this algorithm is *at worst* 20% if the membership of the classes is completely even.

These algorithms are implemented in python 3.

Chapter 5

Results

The result of each algorithm is presented with relative performance against each of the baseline algorithms. Since the two node embeddings node2vec and gat2vec and the graph embedding graph2vec differ in what data they use for the classification task as well as differ in the classification method, they are not compared against each other.

5.1 node2vec

The average result of the node2vec algorithm are shown in table 5.1, together with the standard deviation after 10 runs. The precision, recall and F1-score are presented for each of the classes in addition their micro, macro and weighted averages. The *support* is the average number of data points that is classified to each respective class.

Table 5.1: *The results of the node2vec algorithm averaged from 10 executions, with the standard deviation. This is done with decision tree regression as classification method.*

	Precision	**Recall**	**F1-score**	**Support**
very easy	0.42 ± 0.06	0.30 ± 0.01	0.35 ± 0.03	1203.6 ± 133.7
easy	0.08 ± 0.02	0.39 ± 0.05	0.13 ± 0.03	49.7 ± 14.4
hard	0.60 ± 0.04	0.29 ± 0.01	0.39 ± 0.01	1513.2 ± 131.6
very hard	0.00	0.00	0.00	0.0
impossible	0.01 ± 0.01	0.25 ± 0.07	0.03 ± 0.02	32.5 ± 22.87
Micro	0.30 ± 0.1	0.3 ± 0.01	**0.3** ± 0.01	
Macro	0.22 ± 0.01	0.25 ± 0.02	**0.23** ± 0.01	
Weighted	0.51 ± 0.01	0.30 ± 0.01	**0.37** ± 0.01	

The table shows that the two most populated classes, *hard* and *very easy*, acquire most of the support from the classifier. In fact, about 97% of the support is carried by these classes, whereas less than 60% of the nodes in the data set actually belong to these classes. A normalized confusion matrix is presented in figure 5.1 where the diagonal represents true predicted values.

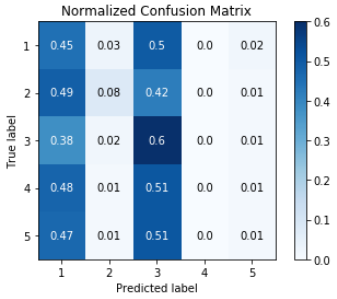

Figure 5.1: *Normalized confusion matrix of the TTC prediction of one of the 10 runs of the node2vec algorithm with the final parameter settings. The numbers corresponds to each class in order, where 1=very easy.*

5.2 gat2vec

The averaged result of the gat2vec algorithm is presented in 5.2, with the standard deviation. Like the node2vec implementation, the largest classes *very easy* and *hard* carry most of the support, about 89%.

Table 5.2: *The results of the gat2vec algorithm, averaged from 10 execution, with the standard deviation.*

	Precision	**Recall**	**F1-score**	**Support**
very easy	0.57 ± 0.01	0.39 ± 0.01	0.46 ± 0.01	1257.5 ± 29.7
easy	0.08 ± 0.02	0.27 ± 0.04	0.12 ± 0.03	67.8 ± 8.8
hard	0.48 ± 0.02	0.29 ± 0.01	0.36 ± 0.01	1241.7 ± 45.3
very hard	0.00	0.19 ± 0.14	0.01 ± 0.00	8.2 ± 4.4
impossible	0.12 ± 0.01	0.29 ± 0.02	0.27 ± 0.01	223.8 ± 17.1
Micro	0.33 ± 0.01	0.33 ± 0.01	**0.33** ± 0.01	
Macro	0.25 ± 0.01	0.28 ± 0.03	**0.26** ± 0.01	
Weighted	0.48 ± 0.01	0.33 ± 0.01	**0.38** ± 0.01	

In general, the weighted average gives a higher score than the micro and macro averaging methods, and micro higher than the macro. The DTC votes for the majority class in each region, and there will be more regions with a majority of the larger classes, which leads to a relatively low score in the smaller classes. As a result, the macro average score that weights all classes equally, is low in comparison.

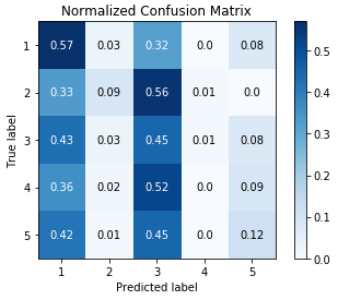

Figure 5.2: *Normalized confusion matrix for the TTC prediction of one of the 10 runs of the gat2vec algorithm, where the diagonal corresponds to correctly predicted instances. The numbers corresponds to each class in order, where 1=very easy.*

The confusion matrix in figure 5.3 shows the predicted labels and how it was classified compared to the true labels.

5.3 graph2vec

The table 5.3 displays the final result for the graph2vec algorithm using the SVC to predict the average TTC values for each graph. Compared to the node embedding algorithms that use the DTC, the support for each class is more evenly distributed. Consequently, the macro average method is nearly on par with with the micro an weighted averaging methods.

Table 5.3: *The result of the TTC prediction classification, using the graph2vec embedding with the support vector classifier.*

	Precision	Recall	F1-score	Support
very easy	0.20	0.27	0.23	15
easy	0.44	0.40	0.42	20
hard	0.53	0.46	0.49	37
very hard	0.19	0.25	0.21	12
impossible	0.40	0.33	0.36	6
Micro	0.38	0.38	**0.38**	
Macro	0.35	0.34	**0.34**	
Weighted	0.41	0.38	**0.39**	

The normalized confusion matrix for the predicted result of the graph2vec algorithm is shown in figure 5.3. Compared to each of the node embedding algorithms, the distribution of the predicted values is more evenly distributed. This is due to the choice of classifier, where DTC in contrast to SVC gives a high weight to the larger classes if the minimal sample split parameter is high.

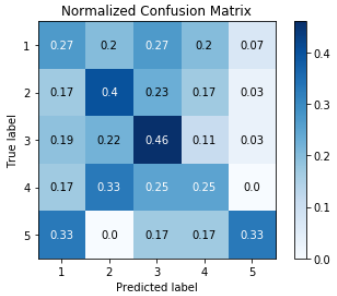

Figure 5.3: *Normalized confusion matrix for the final graph2vec algorithm. The diagonal represent correct predictions, and the integers represents each class in order, where 1=very easy.*

5.4 Baseline Algorithms

Table 5.4 shows the results for both of the baseline algorithms in the node classification case. The random algorithm is presented with its standard deviation. The largest class is *very easy*, hence the constant algorithm predicted all nodes to belong to that class and the *very easy* class precision is 100% as well as the support for that class. The averaged precision, recall and F_1 score for the random baseline method is around 20%, which is as expected with 5 classes.

Table 5.5 displays the results in the graph classification case. Here the largest class is *hard*, which leads to the constant baseline method putting all support on *hard*. The random is evenly distributed with around 20% on each class.

Table 5.4: *Results from the random/constant baseline algorithms in the node classification case. The random algorithm is reported along with its standard deviation. The constant algorithm has* 100*% support on the largest class* very easy*, whereas the random algorithm has its support evenly distributed.*

	Precision	**Recall**	**F1-score**	**Support**
very easy	0.19 ± 0.01 / 1	0.30 ± 0.02 / 0.31	0.24 ± 0.01 / 0.47	552.1 ± 15.1 / 2799
easy	0.19 ± 0.03 / 0	0.08 ± 0.01 / 0	0.11 ± 0.02 / 0	563.7 ± 15.2 / 0
hard	0.19 ± 0.02 / 0	0.26 ± 0.01 / 0	0.22 ± 0.01 / 0	565.8 ± 26.7 / 0
very hard	0.19 ± 0.02 / 0	0.14 ± 0.01 / 0	0.17 ± 0.01 / 0	568.1 ± 28.4 / 0
impossible	0.19 ± 0.02 / 0	0.18 ± 0.01 / 0	0.18 ± 0.01 / 0	549.3 ± 14.1 / 0
Micro	0.19 ± 0.01 / 0.31	0.19 ± 0.01 / 0.31	**0.19** ± 0.01 / **0.31**	
Macro	0.19 ± 0.01 / 0.20	0.19 ± 0.01 / 0.06	**0.18** ± 0.01 / **0.09**	
Weighted	0.19 ± 0.01 / 1.00	0.19 ± 0.01 / 0.31	**0.18** ± 0.01 / **0.47**	

Table 5.5: *Classification report of the uniform/constant baseline methods in graph embedding case. The constant baseline method solely puts support on class hard and the random method is evenly distributed.*

	Precision	**Recall**	**F1-score**	**Support**
very easy	0.18 ± 0.07 / 0	0.18 ± 0.05 / 0	0.18 ± 0.06 / 0	19.8 ± 3.6 / 0
easy	0.20 ± 0.08 / 0	0.26 ± 0.07 / 0	0.22 ± 0.08 / 0	20.1 ± 3.9 / 0
hard	0.19 ± 0.06 / 1	0.32 ± 0.01 / 0.32	0.24 ± 0.06 / 0.48	18.7 ± 3.1 / 100
very hard	0.23 ± 0.12 / 0	0.18 ± 0.10 / 0	0.20 ± 0.11 / 0	20.4 ± 3.9 / 0
impossible	0.28 ± 0.20 / 0	0.07 ± 0.05 / 0	0.11 ± 0.08 / 0	21.0 ± 2.5 / 0
Micro	0.20 ± 0.04 / 0.32	0.20 ± 0.04 / 0.32	**0.20** ± 0.04 / **0.32**	
Macro	0.21 ± 0.05 / 0.20	0.20 ± 0.03 / 0.06	**0.19** ± 0.04 / **0.10**	
Weighted	0.22 ± 0.05 / 1.00	0.20 ± 0.04 / 0.31	**0.19** ± 0.04 / **0.48**	

5.5 Summary

A summary and comparison between the node embedding algorithms and the baseline methods is shown in table figure 5.6. It presents the gain in percent of each of the algorithms compared to the baseline random/constant methods, i.e $gain = algorithm/baseline - 1$. The indices in bold showcases the best result in each category. In two cases the constant baseline method scored the highest, it which case it is in bold. These are the weighted averaged precision score (1.00), and the weighted averaged F_1 score (0.47). In most cases, node2vec and gat2vec shows a superior result compared to the baseline methods and gat2vec slightly so over node2vec.

Table 5.6: *A summary of the predictive score of node embedding algorithms and their relative gain compared to the baseline random/constant methods, using the DTC. The bold indices indicates the best score of each category.*

	Precision			**Recall**			**F1-score**		
	Micro	Macro	Weighted	Micro	Macro	Weighted	Micro	Macro	Weighted
node2vec	0.30	0.22	0.51	0.3	0.25	0.3	0.3	0.23	0.37
gat2vec	**0.33**	**0.25**	0.48	**0.33**	**0.28**	**0.33**	**0.33**	**0.26**	0.38
Baseline R/C	0.19/0.31	0.19/0.20	0.19/**1.00**	0.19/0.31	0.19/0.06	0.19/0.31	0.19/0.31	0.18/0.09	0.18/**0.47**
Gain node2vec [%]	57.9/-3.2	15.8/10.0	**168.4**/-49.0	57.9/-3.2	31.6/316.7	57.9/-3.2	57.9/-3.2	27.8/155.6	105.6/-21.3
Gain gat2vec [%]	**73.7**/**6.5**	**31.6**/**25.0**	152.6/-52.0	**73.7**/**6.5**	**47.4**/**366.7**	**73.68**/**6.5**	**73.7**/**6.5**	**44.4**/**188.9**	**111.11**/-19.1

The result of the class prediction using graph2vec and SVC is compared to the baseline methods in table 5.7. Similar to the node embedding method, the weighted avarage of the constant baseline method gives a high F_1 score and a high weighted score.

Table 5.7: *The predictive scores of the graph classification task using the graph2vec algorithm and SVC, and compared to the baseline random/constant approaches.*

	Precision			**Recall**			**F1-score**		
	Micro	Macro	Weighted	Micro	Macro	Weighted	Micro	Macro	Weighted
graph2vec	**0.38**	**0.35**	0.41	**0.38**	**0.34**	**0.38**	**0.38**	**0.34**	0.39
Guessing R/C	0.20/0.32	0.21/0.20	0.22/**1.00**	0.20/0.32	0.20/0.06	0.20/0.31	0.20/0.32	0.19/0.10	0.19/**0.48**
Gain graph2vec [%]	**90.0**/**18.8**	**66.7**/**75.0**	**86.4**/59.0	**90.0**/**18.75**	**70.0**/**466.7**	**90.0**/**22.6**	**90.0**/**18.9**	**78.9**/**240**	**105.3**/-18.5

Chapter 6

Discussion and Future Work

According to Do et al. [9], classifying high dimensional data with DTC is not optimal. The node embeddings can indeed be seen as high dimensional data, which makes the choice of classifier debatable. Another choice would be the SVM classifier that is used to classify the graph2vec algorithm. However, even with a possible sup-par classification method, the result show that each of the implemented embedding algorithms perform considerably better than the naive baseline algorithm of random guessing. Furthermore, the result displays a slight favor of incorporating attributes (gat2vec) over not doing so (node2vec), which is in line with what one would intuitively think matters to predict the TTC value in the IT-architecture. Nonetheless, there are still some questions to be answered.

In both the node embedding algorithms, node2vec and gat2vec, the training of the skip gram is done completely independently on each graph. As a result, there is a limit on how well these algorithms can perform when trying to carry out a classification as if the training was not independent. In fact, they are not built to be compared between graphs [26], and it is not possible generate embeddings for new nodes in a graph that were not seen during training. On the other hand, the benefit of these algorithms is that they are easily expanded to unseen graphs, i.e the embedding of the unseen graph can be directly compared to the graphs used in training. This is essential, since it is not feasible to train on all possible graphs that can be reached by a reinforcement learning algorithm. However, the disadvantage of training the embeddings of each graph independently is undoubtedly significant, and thus making this solution sub-par. For further analysis, it would be valuable to investigate how the node embeddings can be trained with dependence between the graphs, in such a way that it can be expanded to unseen graphs.

The graph embedding method, graph2vec, does not suffer from the independence issue. It trains the skip gram model on all graphs at the same time, and naturally the embedding of each graph is dependent on the embedding of each of the other graphs. On the other hand, this solution suffers from the inability to generalize to unseen graphs. The graphs that are seen in training are the only graphs in the algorithm's vocabulary. As a result, each new graph introduced requires retraining

of all graphs. Consequently, directly incorporating this algorithm to the reinforcement learning framework is certainly not feasible. However, that does not exclude the possibility that graph2vec could be used with slight alterations. For example, one solution to investigate is to store the weights (embeddings) of existing graphs during the reinforcement learning scheme. Henceforth, when a new graph is reached, expand the dimension of the hidden weight matrix and only update the new weights according to the DBOW[1] algorithm. Another solution is by Niepert et al. [37]. They propose to use convolutional neural networks on a collection of graphs, to learn a function that can be used for classification and regression on unseen graphs. In similar fashion, graphSAGE [15] likewise learns to aggregate features about the neigborhood of a node, which can be generalized to unseen nodes. As a result, these methods do not need to be retrained when a new graph or node is introduced.

The three implemented methods are all static methods, i.e the graphs do not evolve over time. However, if an action by an agent in the RL algorithm is modelled as change of a graph rather than visiting a new graph, a dynamic graph embedding could be a viable solution. Dynamic graph embeddings treat graphs that evolve over time. For example, IGE [48] deals with an with dynamic and attributed graph edges. The DANE [28] initiate the algorithm with a static embedding that preserves both structural proximities and node attributes, and updates it with matrix peturbation theory as it evolves over time. Similarly, the DepthLGP [30] initiates a static node embedding and then generate embeddings of nodes that arrive after learning.

Further, these experiments are made on simulating tool that incorporates less features than the actual securiCAD tool. The importance of choosing an embedding that captures the attributes of each node and edge is likely of even higher importance than what the results of this thesis displays.

[1] Distributed bag of words, explained in the 3.2 chapter.

Chapter 7

Conclusion

Some complications arise when trying to apply vector based machine learning techniques to graph data. In this thesis the specific problem is to represent graph data as the state space in a reinforcement learning algorithm. The solution that is evaluated is to map the graph to a vector space using either a node embedding or a graph embedding, and alternatively using the resulting embedding in the reinforcement learning framework. Three algorithms are implemented, two node embedding and one graph embedding. The first one is the node embedding node2vec, that uses biased random walks to capture neighborhood structures of a node. The second one is the node embedding gat2vec, that introduces a bipartite attribute graph and capture the neighborhood of a node by random walking on both the attribute and the structural graph. The third one is graph2vec, that embeds an entire graph by introducing rooted subgraphs. To evaluate these algorithm the already known TTC value[1] for each node is used. Henceforth, it is made into a classification problem, where the aim is to classify if it is *very easy*, *easy*, *hard*, *very hard*, *impossible* for the attacker to reach the node given the embeddings (or average difficulty for the attacker if we have a graph embedding). The result is then compared to the baseline algorithms constant and random guessing. All three algorithms have a significant improvement in terms of $F1$ score compared to the random baseline algorithm - using the micro average method the improvement is 57.9%, 73.7% and 90.0% respectively. Still, the weighted $F1$ score for the constant baseline algorithm is not beaten. As a result, given that the user has prior knowledge about the distribution of the classes in the data, it is arguable if any of the algorithms is better than constant guessing on the most popular class, depending on how the classification score is evaluated. However, the results show that the static node embedding algorithms and static graph embedding algorithm does indeed capture some information about the TTC value of each node, which is essential for using this solution in a reinforcement learning setup. However, even if these results show improvement over naive approaches, the implementation of the Automatic Designer would nevertheless require an even better performance. A natural next research area is dynamic graph embedding, where the graphs can evolve over time. Another promising area are the algorithms that learn functions of

[1] Time To Compromise, the time it takes for an attacker to reach each node

features of graphs, which can thereafter be applied to unseen nodes and graphs [15, 37].

Bibliography

[1] M. Balasubramanian and E. L. Schwartz. The isomap algorithm and topological stability. *Science*, 295(5552):7–7, 2002.

[2] M. Belkin and P. Niyogi. Laplacian eigenmaps and spectral techniques for embedding and clustering. In *Advances in neural information processing systems*, pages 585–591, 2002.

[3] Y. Bengio, A. Courville, and P. Vincent. Representation learning: A review and new perspectives. *IEEE transactions on pattern analysis and machine intelligence*, 35(8):1798–1828, 2013.

[4] L. Breiman. Random forests. *Machine learning*, 45(1):5–32, 2001.

[5] S. Cao, W. Lu, and Q. Xu. Grarep: Learning graph representations with global structural information. In *Proceedings of the 24th ACM International on Conference on Information and Knowledge Management*, pages 891–900. ACM, 2015.

[6] S. Cao, W. Lu, and Q. Xu. Deep neural networks for learning graph representations. In *AAAI*, pages 1145–1152, 2016.

[7] H. Chen, B. Perozzi, Y. Hu, and S. Skiena. Harp: Hierarchical representation learning for networks. In *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

[8] B. V. Cherkassky, A. V. Goldberg, and T. Radzik. Shortest paths algorithms: Theory and experimental evaluation. *Mathematical programming*, 73(2):129–174, 1996.

[9] T.-N. Do, P. Lenca, S. Lallich, and N.-K. Pham. Classifying very-high-dimensional data with random forests of oblique decision trees. In *EGC*, 2009.

[10] M. Ekstedt, P. Johnson, R. Lagerstrom, D. Gorton, J. Nydrén, and K. Shahzad. Securi cad by foreseeti: A cad tool for enterprise cyber security management. In *2015 IEEE 19th International Enterprise Distributed Object Computing Workshop (EDOCW)*, pages 152–155. IEEE, 2015.

[11] S. Fortin. The graph isomorphism problem. Technical report, Citeseer, 1996.

[12] P. Goyal and E. Ferrara. Gem: A python package for graph embedding methods. *Journal of Open Source Software*, 2018.

[13] P. Goyal and E. Ferrara. Graph embedding techniques, applications, and performance: A survey. *Knowledge-Based Systems*, 2018.

[14] A. Grover and J. Leskovec. node2vec: Scalable feature learning for networks. *CoRR*, abs/1607.00653, 2016.

[15] W. Hamilton, Z. Ying, and J. Leskovec. Inductive representation learning on large graphs. In *Advances in Neural Information Processing Systems*, pages 1024–1034, 2017.

[16] W. L. Hamilton, R. Ying, and J. Leskovec. Representation learning on graphs: Methods and applications. *arXiv preprint arXiv:1709.05584*, 2017.

[17] T. Hastie, R. Tibshirani, and J. Friedman. *The elements of statistical learning: data mining, inference and prediction*. Springer, 2 edition, 2009.

[18] G. E. Hinton and R. R. Salakhutdinov. Reducing the dimensionality of data with neural networks. *science*, 313(5786):504–507, 2006.

[19] H. Holm, M. Korman, and M. Ekstedt. A bayesian network model for likelihood estimations of acquirement of critical software vulnerabilities and exploits. *Information and Software Technology*, 58:304–318, 2015.

[20] J. Jacobsson. Securilang technical documentation, 2017. Edition 1.2.8.

[21] G. James, D. Witten, T. Hastie, and R. Tibshirani. *An Introduction to Statistical Learning: With Applications in R*. Springer Publishing Company, Incorporated, 2014.

[22] R. Lagerström, P. Johnson, and M. Ekstedt. Automatic design of secure enterprise architecture: Work in progress paper. In *Enterprise Distributed Object Computing Workshop (EDOCW), 2017 IEEE 21st International*, pages 65–70. IEEE, 2017.

[23] P. Latouche and F. Rossi. Graphs in machine learning: an introduction. *ArXiv e-prints*, June 2015.

[24] Q. Le and T. Mikolov. Distributed representations of sentences and documents. In *International Conference on Machine Learning*, pages 1188–1196, 2014.

[25] J. Leskovec, W. L. Hamilton, R. Ying, and R. Sosic. Representation learning on networks, part 1: Node embeddings. `http://snap.stanford.edu/proj/embeddings-www/`, 2018. Accessed: 2018-11-14.

[26] J. Leskovec, W. L. Hamilton, R. Ying, and R. Sosic. Representation learning on networks, part 2: Graph neural networks. `http://snap.stanford.edu/proj/embeddings-www/`, 2018. Accessed: 2018-11-14.

[27] J. Leskovec and R. Sosič. Snap: A general-purpose network analysis and graph-mining library. *ACM Transactions on Intelligent Systems and Technology (TIST)*, 8(1):1, 2016.

[28] J. Li, H. Dani, X. Hu, J. Tang, Y. Chang, and H. Liu. Attributed network embedding for learning in a dynamic environment. In *Proceedings of the 2017 ACM on Conference on Information and Knowledge Management*, pages 387–396. ACM, 2017.

[29] J. Liang, P. Jacobs, J. Sun, and S. Parthasarathy. Semi-supervised embedding in attributed networks with outliers. In *Proceedings of the 2018 SIAM International Conference on Data Mining*, pages 153–161. SIAM, 2018.

[30] J. Ma, P. Cui, and W. Zhu. Depthlgp: Learning embeddings of out-of-sample nodes in dynamic networks. In *AAAI 2018*, 2018.

[31] C. McCormick. Word2vec tutoral - the skip-gram model. `http://www.mccormickml.com`, 2016, April 19. Accessed: 2019-02-16.

[32] C. McCormick. Word2vec tutoral part 2 - negative sampling. `http://www.mccormickml.com`, 2017, January 11. Accessed: 2019-02-16.

[33] T. Mikolov, K. Chen, G. Corrado, and J. Dean. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*, 2013.

[34] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean. Distributed representations of words and phrases and their compositionality. In *Advances in neural information processing systems*, pages 3111–3119, 2013.

[35] A. Narayanan, M. Chandramohan, L. Chen, Y. Liu, and S. Saminathan. subgraph2vec: Learning distributed representations of rooted sub-graphs from large graphs. *CoRR*, abs/1606.08928, 2016.

[36] A. Narayanan, M. Chandramohan, R. Venkatesan, L. Chen, Y. Liu, and S. Jaiswal. graph2vec: Learning Distributed Representations of Graphs. *ArXiv e-prints*, July 2017.

[37] M. Niepert, M. Ahmed, and K. Kutzkov. Learning convolutional neural networks for graphs. In *International conference on machine learning*, pages 2014–2023, 2016.

[38] M. Ou, P. Cui, J. Pei, Z. Zhang, and W. Zhu. Asymmetric transitivity preserving graph embedding. In *Proceedings of the 22nd ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 1105–1114. ACM, 2016.

[39] S. Pan, J. Wu, X. Zhu, C. Zhang, and Y. Wang. Tri-party deep network representation. *Network*, 11(9):12, 2016.

[40] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[41] B. Perozzi, R. Al-Rfou, and S. Skiena. Deepwalk: Online learning of social representations. *CoRR*, abs/1403.6652, 2014.

[42] N. Sheikh. Gat2vec. `https://github.com/snash4/GAT2VEC`, 2018.

[43] N. Sheikh, Z. Kefato, and A. Montresor. gat2vec: representation learning for attributed graphs. *Computing*, pages 1–23, 2018.

[44] N. Shervashidze, P. Schweitzer, E. J. van Leeuwen, K. Mehlhorn, and K. M. Borgwardt. Weisfeiler-lehman graph kernels. *J. Mach. Learn. Res.*, 12:2539–2561, Nov. 2011.

[45] A. Taheri, K. Gimpel, and T. Berger-Wolf. Learning graph representations with recurrent neural network autoencoders. *KDD Deep Learning Day*, 2018.

[46] V. Van Asch. Macro-and micro-averaged evaluation measures [[basic draft]]. *Belgium: CLiPS*, 2013.

[47] D. Wang, P. Cui, and W. Zhu. Structural deep network embedding. In *Proceedings of the 22Nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '16, pages 1225–1234, New York, NY, USA, 2016. ACM.

[48] Y. Zhang, Y. Xiong, X. Kong, and Y. Zhu. Learning node embeddings in interaction graphs. In *Proceedings of the 2017 ACM on Conference on Information and Knowledge Management*, pages 397–406. ACM, 2017.

TRITA -SCI-GRU 2019:046