TERRITORI APERTI

FONDO TERRITORI LAVORO E CONOSCENZA CGIL, CISL, UIL

Deliverable

A Reinforcement Learning approach to Anomaly Detection in Temporal Graphs

http://territoriaperti.univaq.it

UNIVERSITÀ DEGLI STUDI DELL'AQUILA

Project Title	:	Territori Aperti

Title of Deliverable	:	A Reinforcement Learning approach to Anomaly Detection in Temporal Graphs
Nature of Deliverable	:	Technical Report
Dissemination level	:	Public
Version	:	3.0
Contractual Delivery Date	:	2021
Actual Delivery Date	:	2021
Contributing WP	:	WP3.2
Author(s)	:	Benedetta Flammini (University of L'Aquila) and Giovanni Stilo (University of L'Aquila)

Abstract

Keyword List

Anomaly Detection, Temporal Graph, Reinforcement Learning

Glossary, acronyms & abbreviations

Item	Description
A.D.	Anomaly Detection
D.N.	Dynamic Networks
R.L.	Reinforcement Learning
T.G.	Temporal Graph

Table Of Contents

List Of Tables

List Of Figures

1 Introduction

Detecting anomalies in data is a fundamental task with a wide range of high-impact applications in different domains, as finance, security, and health. In the past two decades, there has been a growing interest in anomaly detection in data represented as graphs, due to their natural ability to capture and represent real-world relationships. At first, the techniques mainly focus on static graphs, which can capture a single snapshot of the data at a certain time. Thus, to capture the dynamic and constantly evolutionary aspects of the real world, the researchers have shifted their attention to dynamic graphs, which evolve.

The report aims to explore the use of reinforcement learning techniques, in particular those based on dynamic programming, to perform anomaly detection on temporal graphs. Since the report collects the first application of these techniques in this new scenario, we explored its feasibility by testing on a set of small undirected graphs and by using dynamic programming techniques. These choices allow us to obtain a first realistic evaluation of the method's performance and fostering the number of computational resources needed. A chapter will be dedicated to the implementation of the method and the analysis of the results obtained by the experimentation.

The main contributions of the report are:

- proposing the formalization for the first reinforcement learning approach to anomaly detection on temporal graphs, laying the foundations for further developments;
- defining an iter to produce sequences of temporal graphs starting from a graph generative model;
- providing the implementation of the algorithm, that has a good performance;
- defining an experimental protocol to test the goodness of the method;
- extending the proposed approach to an unsupervised setting, trying to generalize as much as possible.

1.1. Report organisation

The report will be organized as follows:

- in Chapter 1 we discuss all the background concepts needed to formalise the anomaly detection in temporal graphs and understand the used reinforcement learning techniques;
- Chapter 2 presents our theoretical framework regarding the application of dynamic programming on temporal graphs to perform anomaly detection;
- Chapter 3 shows the practical implementation of the developed technique, the evaluation protocol that we carried and the achieved results;
- in Chapter 4 we summarise the research that has been conducted, the learned lesson and the possible future developments.

1.2. Remarks

We like to note that the presented work represents an initial exploration of a wider and more complete yet complex project. Thus, the main objective of this report is formalising the anomaly detection problem on the dynamic graphs considering a reinforcement learning approach. For this reason, we explore a simple but well-defined application in controlled experimental settings which allows us to evaluate consistently the benefit and the drawbacks of this approach. We highlight that the general aim of the report must be framed into a more complete project where the objective is to create an efficient and robust but yet flexible detector that can be practically used in detecting real-time anomalies.

2 Background

This chapter presents the background knowledge, focusing on the existing techniques, for both anomaly detection in static/temporal graphs and reinforcement learning. Moreover, a focus on graph generation techniques is provided. The aim is to provide a general overview and a basis for the comprehension of the report.

Data objects cannot always be treated as independent points lying in a multi-dimensional space, but they may show inter-dependencies that need to be considered. Graphs provide a powerful instrument for capturing these relations among inter-dependent data objects. Four main reasons make anomaly detection on graphs fundamental, as explained in [3]:

- **Inter-dependent nature of data:** as mentioned above, data are often related to each other, and this represents an important factor to consider when detecting anomalies.
- **Powerful representation:** graphs are naturally able to represent inter-dependencies through the introduction of links between the related objects. Besides, graphs enable to represent more complex structures thanks to node and edge attributes.
- **Relational nature of problem domains:** the anomalies themselves could show to be related one another, and in order to understand it a graph representation is more suitable.
- **Robust machinery:** graphs may serve as more adversarially-robust tools, since fitting in a network without having the complete knowledge of it is hard.

It is not possible to give a unique definition of the anomaly detection problem, since it can be defined in various way according to the specific context or application. Nevertheless, a general definition for the graph anomaly problem is presented in [3]:

[General Graph Anomaly Detection Problem] Given a (plain/attributed, static/dynamic) graph database, find the graph objects (nodes, edges or substructures) that are rare and that differ significantly from the majority of the reference objects in the graph. Usually, a graph object is labelled as anomalous if its rarity score exceeds a given threshold.

2.1. Anomaly Detection in Static Graphs

In this section we will treat the anomaly detection problem in static graphs, where the main objective is to spot anomalous network entities given the entire graph structure. In [3] this problem is defined as:

[Static graph anomaly detection problem] Given the snapshot of a (plain or attributed) graph database, find the nodes and/or edges and/or substructures that are “few and different” or deviate significantly from the patterns observed in the graph.

We will study anomaly detection in static graphs under two different settings: attributed graphs, where nodes and/or edges have features associated, and plain graphs, where only the structure is taken into account.

2.1.1. Anomaly Detection in static plain graphs

In a plain graph, the only information is its structure, so the methods of anomaly detection rely solely on the structure to spot anomalies and find patterns. We can divide the patterns in structure-based patterns and community-based patterns.

The structure-based methods in turn can be divided in feature based and proximity-based.

The feature-based approaches exploit the graph structure to extract structural graph-centric features for performing outlier detection. These methods transform the graph anomaly detection problem into the outlier detection problem. The features that can be extracted are of various types: node-level features, as in and out-degree, betweenness centrality, closeness centrality; or node-group-level features, such as density, modularity; or global measures, such as the number of connected components, global clustering coefficient, average node degree and so on.

The main idea of the proximity-based methods is to exploit the graph structure to measure the closeness of the objects in the graph, assuming that close objects are likely to belong to the same class. Some of the most widely used graph-closeness measures are based on random walks and their extensions. A prominent example in this sense is the PageRank [4], one of the most popular algorithms based on random walks, whose aim is to measure the importance of the nodes in a graph.

Community or cluster-based methods rely on finding densely connected groups of nodes in the graph, and spot those nodes and/or edges that have connections across communities. In this case, nodes/edges can be considered anomalous if they do not belong to one particular community. An example of real-world application includes publication networks, where the aim is to find those papers that authors from different research communities write. Clearly, the main tasks are how to find the communities of a given node and how to quantify the level of the given node to be a bridge node [5].

2.1.2. Anomaly Detection in static attributed graphs

Certain types of data allow having a richer graph representation, in which nodes and/or edges have attributes. For example, it is possible to think of social networks in which the interests of each user (node) are expressed.

For these graphs, it is possible to exploit both the structure and the attributes to detect anomalies. These methods can be grouped into structure-based, community-based, and relational learning-based methods.

One of the earliest works on attributed graph anomaly detection [6] highlights two problems: finding unusual substructures in a given graph and finding unusual subgraphs given a set of subgraphs. Structure-based approaches aim to identify substructures in the structurally rare graphs, both from the point of connectivity and attributes. The idea is to look at the most frequent substructures, considering them as normal.

The community-based methods aim to identify those nodes in a graph whose attribute values deviate significantly from the other community members they belong to. Both structural aspects and attributes aspects are considered when identifying communities. Due to this dual aspect, some methods aim to detect outliers and communities simultaneously, while others perform first attributed graph clustering and then outlier detection.

Relational learning-based methods include network-based classification algorithms, whose main idea is to exploit the relationships between objects to assign them into classes (usually normal and anomalous). Classification is the problem of labeling data instances according to their observed attributes: anomaly detection can be formulated as a classification problem. When the labeled data is reasonable enough, fully supervised classification can be employed; when labeled data is scarce but available, semi-supervised classification can be used.

In traditional machine learning approaches, the instances are assumed to be identical independent distributed, ignoring the dependencies. Relational classification instead aims at labeling objects that carry important information in their relationships. Generally, when labeling a node, these methods exploit the class labels of its neighbors, the node attributes, and/or the attributes of the node's neighbors.

Relational classification methods can be categorized into local and global. The local algorithms focus on building a local predictive model for classifying a node, often using iterative inference procedures to classify unlabeled objects. Global algorithms, instead, aim at defining a global formulation of class dependencies.

2.2. Anomaly Detection in Temporal Graphs

For what concerns dynamic graphs, before illustrating the anomaly detection problem, we can define it as in [1]: [Dynamic Graph] We can define a graph series $\mathbb{G}$ (or dynamic graph) as an ordered set of graphs with a fixed number of time steps. Formally, $\mathbb{G} = \{G_t\}_{t=1}^{T}$, where T is the total number of graphs, and $G_t = (V_t, E_t \subseteq (V_t \times V_t))$, with V_t the vertex set at time step t and E_t the edge set at time step t. Graph series where $t \to \infty$ are called graph streams.

The anomaly detection problem on dynamic graphs can be summarized as follows [3]: [Dynamic Graph Anomaly Detection Problem] Given a sequence of plain/attributed graphs, find (i) the timestamps that correspond to a change or event, as well as (ii) the top-k nodes, edges, or parts of the graphs that contribute most to the change (attribution).

According to the application domain, the requirements of the algorithm vary, but there are some properties that are generally desirable:

- **Scalability:** due to the size and volume of the graphs produced, ideally the algorithms should be linear or sub-linear with respect to the size of the input graphs. In the dynamic setting it is also preferable that the algorithm is linear with respect to the size of the update of the input graphs.
- **Sensitivity to structural and contextual changes:** the algorithms should be able to distinguish structural differences among the considered input graphs as well as changes in other properties of the graphs, such as labels.
- **Awareness of the importance of change:** the algorithms should be aware of the type and extent of the change, giving more importance to changes in fundamental nodes, edges or other graph attributes, rather than changes in less important structures.

2.2.1. Types of anomalies

Following [1], we identify and formalize four types of anomalies that occur in dynamic networks. Since the graphs are dynamic, vertices and edges can be removed at every time step: for simplicity, we will assume that vertex/edge correspondence is preserved across different time steps thanks to unique labelling of vertices/edges. We now describe the possible four types of anomalies.

Anomalous vertices

Anomalous vertex detection aims to find those vertices that have an irregular evolution if compared to the other vertices in the graph. Generally, each method defines a scoring function for evaluating the vertices behavior time step to time step. The scoring function depends on the particular application. The set of anomalous vertices is defined as:

[Anomalous vertices] Given $\mathbb{G}$, the total vertex set $V = \bigcup_{t=1}^{T} V_t$ and a specified scoring function $f : V \to \mathbb{R}$, the set of anomalous vertices $V' \subseteq V$ is a vertex set such that $\forall$ v' $\in V'$, $|f(v') - \hat{f}| > c_0$, where $\hat{f}$ is a summary statistic of the scores f(v), $\forall$ v $\in V$.

Differently from static graphs, where the single snapshot allows only intragraph comparison, dynamic graphs allow the temporal dynamics of the vertices to be included, introducing new types of anomalies.

This type of anomaly detection is typical for identifying vertices that contribute the most to a discovered event, and for observing the shifts in community involvement.

Anomalous edges

Similarly to vertex detection, edge detection aims to find those edges that have an irregular behavior with respect to the other edges. Also in this case, methods employ an application-dependent scoring function to rate the level of anomalousness of the edges. Anomalous edges detection can be defined as:

[Anomalous edges] Given $\mathbb{G}$, the total edge set $E = \bigcup_{t=1}^{T} E_t$, and a specified scoring function $f : E \to \mathbb{R}$, the set of anomalous edges $E' \subseteq E$ is an edge set such that $\forall$ e' $\in E'$, $|f(e') - \hat{f}| > c_0$, where $\hat{f}$ is a summary statistic of the scores f(e), $\forall$ e $\in E$.

In a static graph it is possible to find the distribution of the edge weights, consequently assigning a score to each edge according to the probability of its weight. In dynamic graphs two new types of irregular edge evolution can be found: (1) abnormal edge evolution, where the weight of a single edge has irregular fluctuations and peaks, and (2) appearance of edges between two vertices that typically are not connected or that belong to different communities.

Applications of this type of anomaly detection include finding unlikely social interactions and identifying anomalous traffic patterns in a graph where vertices are road intersections and edges are roads.

Anomalous subgraphs

Differently from vertices or edges, it is not possible to enumerate every possible subgraph in each single graph, so a different approach is required. Usually, the subgraphs that are identified and tracked are constrained, for example, to those found through community detection methods. Consequently, matching algorithms are required to keep track of the evolution of the subgraphs.

Once a set of subgraphs has been determined, it is possible to perform intragraph comparison or intertime point comparison to assign scores to each subgraph, measuring the change in the subgraphs between adjacent steps. Clearly, typical subgraph comparison methods must be extended to incorporate the additional information coming from the dynamics of the network. For this reason, instead of finding structures that are unique in a single graph, the idea is finding also structures that are unique to a single graph in a series of graphs. These types of anomalies are unique to dynamic networks, and include communities that split, merge, disappear and so on. Two examples in this sense are the shrunken communities, that is when a single community loses a significant number of vertices between time steps, and split communities, when a community divides into several distinct communities.

[Anomalous subgraphs] Given $\mathbb{G}$, a subgraph set $H = \bigcup_{t=1}^{T} H_t$, and a specified scoring function $f : H \to \mathbb{R}$, the set of anomalous subgraphs $H' \subseteq H$ is an subgraph set such that $\forall$ h' $\in H'$, $|f(h') - \hat{f}| > c_0$, where $\hat{h}$ is a summary statistic of the scores f(h), $\forall$ h $\in H$.

Clearly, looking also at the definition, it emerges that what constitutes an anomalous subgraph depends on the application context.

Event and change detection

These two types of anomalies are exclusively found in dynamic graphs. Event detection has the object of identifying time points that are significantly different from the rest, that is isolated points in time in which the graph is different from the graphs at the previous and following time steps. [Event detection] Given $\mathbb{G}$ and a scoring function $f : G_t \to \mathbb{R}$, an event is defined as a time point t, such that $|f(G_t) - f(G_{t-1})| > c_0$ and $|f(G_t) - f(G_{t+1})| > c_0$.

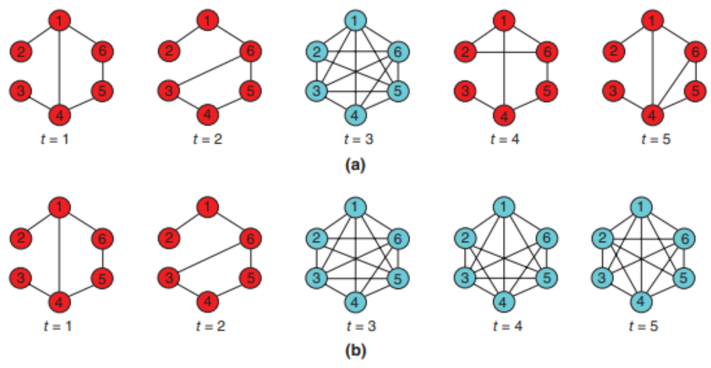


Figure 2.1: The graph series (a) is an example of event detection, the graph series (b) is an example of change detection; picture taken from [1].

The task of finding the cause of the event is not part of event detection, but once the time points for events have been identified it is possible to find the potential causes applying techniques for anomaly detection in static graphs. Event detection has a wide variety of applications, from biology to finding frames in a video that are unlike the others.

Change detection is somehow complementary to event detection: while events represent isolated incidents, change points mark a point in time where the entire behavior of the graph changes and is maintained.

[Change detection] Given $\mathbb{G}$ and a scoring function $f : G_t \to \mathbb{R}$, an change is defined as a time point t, such that $|f(G_t) - f(G_{t-1})| > c_0$ and $|f(G_t) - f(G_{t+1})| \leq c_0$.

One of the most popular applications of change detection is in networks modeling human interactions, such as communication and co-authorship networks.

In order to understand better the difference between change and event detection, we can refer to the following picture taken from [1]:

In event detection the aim is to find a graph that differs from those that precede and succeed it, an isolated case, while in change detection the objective is to spot a shift in the graph series that is maintained.

2.2.2. Methods

The abundance of time-evolving graphs in the recent years has led to a great interest in them, so many researches have been carried on in this field.

Most of the methods of the last two decades have a common two-stage methodology: in the first stage, the aim is to generate data-specific features, and this is done through a mapping that goes from the domain-specific representation (the graph) to a common data representation (a vector of real numbers).

$$\textit{Stage 1} \implies \Omega : G_t \to [\mathbb{R}^d]^t, \quad \forall \quad G_t \in \mathbb{G}, \tag{2.1}$$

where $\mathbb{G}$ is the graph series, G_t is the graph at the time step t, d is the number of feature dimension, $[\mathbb{R}^d]^t$ denotes the feature vector of real numbers for the graph at time t.

The second stage consists in the application of an anomaly detector method to the output of stage one, considering optionally also some historical data. The mapping indicates whether the data is anomalous or not.

$$\textit{Stage 2} \implies f : \{[\mathbb{R}^d]^j, [\mathbb{R}^d]^t\} \to [0,1], \quad j \in [t-k, t-1], \tag{2.2}$$

where $[\mathbb{R}^d]^j$ represents the history and k is arbitrarily chosen. This stage can be generalized by mapping to [0,1] instead of {0,1}, representing the probability that the given data is anomalous.

Focusing the attention on anomaly detection in temporal graphs, most of the methods are interested in finding the best mapping from dynamic graphs to a vector of real numbers, and then applying existing anomaly detectors.

It is possible to categorize the methods in:

- community detection;
- compression;
- matrix/tensor decomposition;
- distance measures;
- probabilistic models.

Community-based detection methods keep track of the evolution of communities over time. The data-specific features are usually derived using the community structure of the graphs, what differs lies in the aspects of the community structure analyzed and in the definition of community used. In fact, it is possible to distinguish between soft communities, where each vertex has a probability of belonging to each community, and disjoint communities, where each vertex belongs to one community at most. Moreover, according to how the evolution of communities is viewed, it is possible to detect different types of anomalies:

- **Vertex detection:** vertices that belong to the same community are expected to behave in a similar way. So, if at consecutive time steps a vertices has a behavior that is different from the other vertices in the community (for example, many new edges are added), that vertex is considered to be anomalous. Referring to soft communities, vertices whose change in belongingness is different from the average change of the vertices in the same community are said to be evolutionary community outliers.
- **Subgraph detection:** the aim is to found entire subgraphs that behave abnormally by looking at the behavior of communities themselves. In [7], six different types of community-based anomalies are distinguished: shrink, grow, merge, split, born and vanish. A graph at time t is compared to the graphs at time $t-1$ and $t+1$, usually through community representatives in order to reduce the computation required.
- **Change detection:** changes are detected by partitioning the streaming graphs into segments based on the similarity of their communities. Partitioning can be achieved in many ways; when the partitioning of the new graph is much different from the one of the current segment, a new segment begins, so the beginning of each segment represents a detected change.

Many compression methods are based on the Minimum Description Length principle. MDL principle is a powerful method of inductive inference: it holds that the best explanation, given a limited set of observed data, is the one that permits the greatest compression of the data. The idea is to exploit patterns and regularity to achieve a compact graph representation. In order to apply this principle to graphs, the adjacency matrix of the graph is seen as a single binary string, flattened in row or column major order. If the rows and the column can be rearranged in a way such that the entropy of the string representation is minimized, then the compression cost is minimized. In this case anomalies are those graphs or substructures that inhibit compression. Referring to specific types of anomalies:

- **Edge detection:** an edge is considered anomalous if the compression of a subgraph has higher encoding cost when the edge is included than when it is omitted.
- **Change detection:** the idea is that consecutive time steps that are similar can be grouped together leading to low compression cost. When the compression cost increases, it means that the new time step differs significantly from the previous ones, representing a change.

In matrix/tensor decomposition techniques the set of graphs is represented as a tensor, and then tensor factorization or dimensionality reduction is performed. In general, for modeling a dynamic graph as a tensor, the method is to create a dimension for each graph aspect of interest. The idea is search and exploit patterns and regularity in the data. All of the data-specific features are derived from the decomposition of a matrix or a tensor, but the main differences among the decomposition-based methods are whether they use a matrix or a higher order tensor, how the tensor is constructed, and the method of decomposition.

Looking at the specific types of anomalies:

- **Vertex detection:** matrix decomposition is used to obtain activity vectors per vertex. A vertex is considered anomalous if its activity changes significantly between consecutive time steps.
- **Event detection:** there are to principal approaches to perform event detection. The first approach is to exploit the reconstruction error: tensor decomposition approximates the original data in a reduced dimensionality, and the reconstruction error is an indicator of how good the approximation is. Elements in the tensor that have a high reconstruction error reveal anomalous vertices, subgraphs or event, because they do not exhibit the typical behavior of the elements surrounding them. The second approach is based on the tracking of the evolution in time of singular values and vectors, in order to detect significant changes.
- **Change detection:** the activity vector of a graph, $u(t)$, is the principal component, the left singular vector corresponding to the largest singular value obtained by performing Singular Value Decomposition on the weighted adjacency matrix. A change point is when an activity vector is substantially different from the ‘normal activity’ vector, which is derived from previous activity vectors.

Using the notion of distance as a metric to measure change is a natural and widely used concept. Two objects that have a small distance in a given metric can be considered similar. Usually, the metrics used in graphs regard structural features: once the summary metrics are found for each graph, the difference or similarity can be calculated. Algorithms differentiates themselves for the metrics chosen and for the methods used to determine the anomalous values and corresponding graphs. In particular, restricting to the different types of anomalies:

- **Edge detection:** if the evolution of some edge attribute differs from the normal evolution, then the corresponding edge is characterized as anomalous. For example, it is possible to study how the weights of the edges evolve or the persistence and frequency of edges.
- **Subgraph detection:** a subgraph with many anomalous edges is considered anomalous.
- **Event detection:** in order to detect events, the idea is to select a function $f(G_i, G_j)$ that measure the distance between two graphs and construct a time series applying the function on consecutive time steps in the graph series. Anomalous values can then be extracted in various ways using a number of different heuristics. Extracting features from the graphs is a straightforward way to create a summary of the graph, its signature. It is possible to distinguish between local features, that are specific to a single vertex and its ego-net, and global features, derived from the graph as a whole. Local features of every vertex can be aggregated into a single signature vector that describes the graph. As an alternative to extracting multiple features, it is also possible to compute pairwise vertex affinity scores, a measure of how much each vertex influences another vertex, or to look at the structural differences between graphs in order to identify the magnitude of change. Events can also be detected using the time series of robustness values for each graph: robustness is a measure of the connectivity of the graph, so a graph with high robustness will retain the same general structure and connectivity even when some edges or vertices are removed. An event happens when the robustness value changes significantly.

Probabilistic methods have a foundation in probability theory, distributions and statistics: they build a model of what is considered normal and label as anomalous the deviations from this model. The methods differs one another for the type of model used, how it is constructed, and the way in which outliers are determined. Usually, the models are built using past data. Anomaly detectors in probabilistic methods do not always perform a strict mapping from features to [0,1] (anomalous or not), but often provide a probability that the structure is anomalous.

- **Vertex detection:** to detect anomalous vertices there is the approach of building scan statistics time series and detecting points that are several stand deviation away from the mean, and the approach of vertex classification. In a graph, a scan statistic can be considered as the maximum of a graph invariant feature.
- **Edge detection:** edges are modelled using a counting process, and edges that deviate from the model by a statistical amount are labelled anomalous.
- **Subgraph detection:** fixed subgraphs, multi-graphs, and cumulative graphs are used to construct models on the expected behaviors. When there is a deviation from the model, there is an anomalous subgraph.
- Event detection: deviations from the models of the graph likelihood or the distribution of the eigenvalues reveal when an event occurs.

Due to the wide range of algorithms, it can be difficult to decide which one to use. The choice depends on the type of graph that is analyzed, the type of anomalies to find, the size of the graphs. However, using multiple methods in parallel or in conjunction may yield to better results compared to using a single method.

2.2.3. Challenges

There are many challenges associated with anomaly detection and attribution of the anomalies, we can divide them in three sub-groups as in [3]: data-specific challenges, problem-specific challenges and graph-specific challenges.

Data-specific challenges are the same encountered when working with big data, so volume, velocity and variety of datasets. The same challenges generalize to graph data as well. We can list:

- **Scale and dynamics:** thanks to technology developments, now it is much easier to collect and analyze large datasets. Not only is the size of data in tera-bytes to peta-bytes, but also the rate at which it arrives is huge.
- **Complexity:** in addition to data size and dynamicity, the datasets are rich and complex in content. As a result, graph representations are very complex due to the incorporation of the additional information available, and nodes and edges can have a long list of attributes associated.

Among the problem-specific challenges, we can find:

- **Lack and noise of labels:** data often come without class labels, that is, there is not a ground truth about whether data instances are anomalous or not. Manual labelling the data is usually not feasible due to the size. Moreover, even when data have labels, they are subject to noises and errors. For these reasons, supervised machine learning algorithms are not very appropriate for the task of anomaly detection.
- **Class imbalance and asymmetric error:** since anomalies are rare events by definition, data have an unbalanced nature and only a very small fraction of data is expected to be abnormal. Besides, the cost of mislabelling an instance depends on the application, and it could be hard to estimate beforehand.

- **Novel anomalies:** especially in the fraud detection domain, when fraudsters understand how the detect algorithms work, they try to change their techniques in order to bypass them. For this reason, algorithms should be adaptive to changing over time.
- **Explaining the anomalies:** in the post-detection phase, there is the additional challenge of explaining anomalies. This involves finding out the root cause of the anomaly and understanding the 'how' and the 'why'. Most of the existing detection techniques do not perform this step, emitting results that are difficult to interpret.

All of the above challenges associated with the anomaly detection problem generalize to graph data, but graph-based anomaly detection has additional challenges as well:

- **Inter-dependent objects:** due to the relation nature of data, it is challenging to quantify the anomalousness of graph objects. In traditional outlier detection, the object are data points treated as independent and identical distributed, while the objects in graph data have correlations. For this reason, when searching for anomalies, these associations need to be taken into account.
- **Variety of definitions:** as we have seen, the definitions of anomalies are much more diverse than in traditional outlier detection, and this surely represents an additional challenge.
- **Size of search space:** when performing anomaly detection on graphs, the main challenge is that the search space is huge. Usually it is not possible to proceed with the enumeration of possible substructures, which makes the anomaly detection problem even harder. Moreover, the search space is enlarged when the graphs are attributed. As a result, graph-based anomaly detection algorithms need to be designed for effectiveness, efficiency, and scalability.

2.3. Reinforcement Learning

Reinforcement Learning is one of the main components of the report. This section is dedicated to the introduction and description of Reinforcement Learning techniques and the elements involved, as illustrated in [2].

When we think about the nature of learning, the first thing that comes to mind is that we learn by interacting with our environment. This kind of connection gives information about cause and effects and consequences and actions. Reinforcement learning can be considered a computational approach to learning from interaction, focused on goal-directed learning interaction. It consists in learning what to do to maximize a numerical reward signal. The learner is not told which actions to take; instead, it must discover which actions yield the most reward by trying them. Clearly, actions may affect the immediate reward and the next situations and thus the subsequent rewards. Trial-and-error and delayed rewards are two distinguishing features of reinforcement learning.

Reinforcement learning is different from supervised learning because the latter is learning from a training set of labeled examples provided by an external supervisor. The object is to generalize the responses so that the system acts correctly in situations not present in the training set. This kind of learning is not appropriate for learning from interaction because, in interactive problems, it is usually impractical to obtain examples of desired behaviors that are both correct and representative of the situations in which the agent has to act. Reinforcement learning is also different from unsupervised learning, typically about finding structures hidden in collections of unlabelled data. Uncovering structure in an agent experience can be useful, but it does not address the reinforcement learning problem. We, therefore, consider reinforcement learning to be a third machine learning paradigm.

One of the challenges that arise in reinforcement learning and not in other kinds of learning is the trade-off between exploration and exploitation: the agent has to exploit what it has already experienced in order to obtain a reward, but it also has to explore in order to make better action selections in the future. Another key feature of reinforcement learning is that it explicitly considers the whole problem of a goal-directed agent interacting with an uncertain environment. This is in contrast to many approaches that consider sub-problems without addressing how they might fit into a larger problem.

Beyond the agent and the environment, one can identify four main elements of a reinforcement learning system:

- **Policy:** it defines the learning agent's way of behaving at a given time. It is a mapping from perceived states of the environment to actions to be taken when in those states. In some cases, the policy may be a simple function or lookup table, whether in others it may involve extensive computation, such as a search process. It is important to distinguish between two types of policies: stochastic and deterministic. In a deterministic policy the mapping is rigid, that is given a state there is just one action to be performed according to the policy. We denote with $\pi(s)$ the action taken in state s under the deterministic policy π. In a stochastic setting, given a certain state, the policy does not return a single action but a set of actions that could be performed from that state each with a certain probability. We denote with $\pi(a|s)$ the probability of taking action a in state s under the stochastic policy π.
- **Reward signal:** it defines the goal of a reinforcement learning problem on each step. The environment sends to the reinforcement learning agent a single number called the reward: the agent's objective is to maximize the total reward it receives over the long run. The reward signal thus defines what are the good and bad events for the agent.
- **Value function:** it specifies what is good in the long run. Roughly speaking, the value of a state is the total amount of reward an agent can expect to accumulate over the future, starting from that state. Whereas rewards determine the immediate desirability of environmental states, values indicate the long-term desirability of states after taking into account the states that are likely to follow and the rewards available in those states. We seek actions that bring about states of highest value, not highest reward, because these actions obtain the greatest amount of reward in the long run.
- **Model of the environment:** it is something that mimics the behavior of the environment, or more generally, that allows inferences to be made about how the environment will behave. Models are made for planning, by which we mean any way of deciding on a course of action by considering possible future situations before they are actually experienced. Methods for solving reinforcement learning problems that use models and planning are called model-based methods, as opposed to model-free methods, that are explicitly trial-and-error learners.

2.3.1. Markov Decision Processes

We formalize reinforcement learning using ideas from dynamical systems theory, specifically as the optimal control of incompletely known Markov decision processes. The basic idea is to capture the most important aspects of the real problem facing a learning agent interacting over time with its environment to achieve a goal. A learning agent must sense the state of its environment to some extent and take actions that affect the state. The agent must also have a goal or goals related to the states of the environment.

Markov decision processes are a classical formalization of sequential decision making, where actions influence immediate rewards and subsequent situations and, through those, future rewards. Thus, MDPs involves the need to trade-off immediate and delayed rewards.

The learner is called an agent; the thing it interacts with, comprising everything outside the agent, is the environment. These interact continually, the agent selecting actions and the environment responding to these actions and presenting new situations to the agent. The environment also gives rise to rewards. More specifically, the agent and the environment interact at each step of a sequence of discrete-time steps. At each time step t, the agent receives some representation of the environment's state, $S_t \in S$ with S space of states, and on that basis selects an action, $A_t \in A$ with A space of actions. One time step later, in part as a consequence of the action, the agent receives a numerical reward, $R_{t+1} \in R \subset \mathbb{R}$ with R space of rewards, and finds itself in a new state S_{t+1}. The MDP and agent together give rise to a sequence or trajectory that begins like this:

$$S_0, A_0, R_1, S_1, A_1, R_2, S_2, A_2, R_3, ...$$

In a finite MDP, the set of states, actions, and rewards all have a finite number of elements. In this case, the random variables S_t and R_t have well-defined discrete probability distributions depending only on the precedent state and action. This means that there is a probability of those values occurring at time t given particular values of the preceding state and action:

$$p(s', r|s, a) = P\{S_t = s', R_t = r|S_{t-1} = s, A_{t-1} = a\} \tag{2.3}$$

for all s, $s' \in S$, $r \in R$ and $a \in A(s)$.

The function $p : S \times R \times S \times A \to [0, 1]$ defines the dynamics of the MDP, it is an ordinary deterministic function of four arguments. It specifies a probability distribution for each choice of s and a, such that:

$$\sum_{s' \in S} \sum_{r \in R} p(s', r|s, a) = 1 \tag{2.4}$$

for all $s \in S$, $a \in A(s)$.

In a Markov decision process, the probabilities given by p completely characterize the environment's dynamics and enable to compute anything else one might want to know about the environment. The state must include information about all aspects of the past agent-environment interaction that make a difference for the future. If it does, then the state is said to have the Markov property.

In reinforcement learning, the purpose or goal of the agent is formalized in terms of a special signal, called the reward, passing from the environment to the agent. At each time step, a reward is a simple number $R_t \in \mathbb{R}$. Informally, the agent's goal is to maximize the total amount of rewards it receives. Although formulating goals in terms of reward signals may appear limiting, it has proved to be very flexible in practice. The reward we set up truly must indicate what we want to be accomplished: communicating to the agent what we want to achieve, not how we want it achieved.

In general, we seek to maximize the expected return, where the return, denoted as G_t, is defined as some specific function of the reward sequence. In the simplest case, the return is the sum of the rewards:

$$G_t = R_{t+1} + R_{t+2} + R_{t+3} + ... + R_T \tag{2.5}$$

where T is a final step.

This approach makes sense in applications where there is a natural notion of the final time step when the agent-environment interaction breaks naturally into sub-sequences, which we call episodes. Each episode ends in a special state called terminal state, followed by a reset to a standard starting or a sample from a standard distribution of starting states. The next episode begins independently of how the previous one ended. Thus, the episodes can all end in the same terminal state, with different rewards for different outcomes. Tasks with episodes of this kind are called episodic tasks.

On the other hand, in many cases, the agent-environment interaction does not break naturally into identifiable episodes but goes on continually without limit: we call these continuing tasks. The return formulation is problematic for continuing tasks because the final time step would be $T = \infty$, and the return could easily be infinite. We need to modify the definition: the additional concept is that of discounts. According to this approach, the agent tries to select actions to maximize the expected discounted return:

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ... + R_T = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \tag{2.6}$$

where $\gamma \in [0, 1]$ is a parameter called discount factor or rate. It determines the present value of future rewards. If $\gamma < 1$, the infinite sum has a finite value as long as the reward sequence $\{R_k\}$ is bounded. Suppose $\gamma = 0$; the agent is concerned only with maximizing immediate rewards. As γ approaches 1, the return objective takes future rewards into account more strongly: the agent becomes more far-sighted.

Almost all reinforcement learning algorithms include estimating value functions. Value functions are defined as particular ways of acting, called policies. Formally, a policy is a mapping from states to probabilities of selecting each possible action. If the agent is following a policy π at time t, then $\pi(a|s)$ is the probability that $A_t = a$ if $S_t = s$.

The value function of a state s under a policy π, denoted $v_\pi(s)$, is the expected return when starting in s and following π thereafter. For MDPs, we can define v_π formally by:

$$\begin{aligned} v_\pi(s) &= \mathbb{E}_\pi[G_t|S_t = s] \\ &= \mathbb{E}_\pi[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1}|S_t = s] \end{aligned} \tag{2.7}$$

for all $s \in S$. We call the function v_π the state-value function for policy π.

Similarly, we define the value of taking action a in state s under policy π, denoted $q(s, a)$, as the expected return starting from s, taking the action a, and thereafter following policy π:

$$\begin{aligned} q(s,a) &= \mathbb{E}_\pi[G_t|S_t = s, A_t = a] \\ &= \mathbb{E}_\pi[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1}|S_t = s, A_t = a] \end{aligned} \tag{2.8}$$

We call $q_\pi(s, a)$ the action-value function for policy π. The value functions v_π and q_π can be estimated from experience. A fundamental property is that for any policy π and any state s, the following consistency condition holds between the value of s and the value of its possible successor states:

$$\begin{aligned} v_\pi(s) &= \mathbb{E}_\pi[G_t|S_t = s] \\ &= \mathbb{E}_\pi[R_{t+1} + \gamma G_{t+1}|S_t = s] \\ &= \sum_a \pi(a|s) \sum_{s'} \sum_r p(s', r|s, a)[r + \gamma \mathbb{E}_\pi[G_{t+1}|S_{t+1} = s']] \\ &= \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a)[r + \gamma v_\pi(s')] \end{aligned} \tag{2.9}$$

The final expression can be read easily as an expected value. It is a sum overall values of the three variables a, s', and r. For each triple, we compute its probability $\pi(a|s)p(s', r|s, a)$, weight the quantity in brackets by that probability, then sum over all the possibilities to get an expected value. The Bellman equation for v_π expresses the relationship between the value of a state and the values of its successor states.

For finite MDPs, we can precisely define an optimal policy in the following way. Value functions define a partial ordering overt policies: a policy π is defined to better or equal to a policy π' if its expected return is greater than or equal to that of π' for all states:

$$\pi \geq \pi' \iff v_\pi(s) \geq v_{\pi'}(s) \quad \forall s \in S \tag{2.10}$$

There is always at least one policy that is better than or equal to all other policies: this is an optimal policy. Although there may be more than one, we denote all the optimal policies by π_*. They share the same state-value function, called the optimal state-value function v_*, and defined as $v_*(s) = max_\pi v_\pi(s)$ for all $s \in S$. Optimal policies also share the same optimal action-value function, q_*, defined $q_*(s, a) = max_\pi q_\pi(s, a)$.

For the state-action pair (s, a), this function gives the expected return for taking action a in state s and thereafter following an optimal policy. Thus, we can write q_* in terms of v_* as follows:

$$q_*(s,a) = \mathbb{E}[R_{t+1} + \gamma v_*(s_{t+1})|S_t = s, A_t = a]. \tag{2.11}$$

Since v_* is the value function for a policy, it must satisfy the Bellman equation, in this case the Bellman optimality equation. Intuitively, it expresses the fact that the value of a state under an optimal policy must equal the expected return for the best action from that state:

$$\begin{aligned} v_*(s) &= max_{a\in A} q_{\pi^*}(s,a) \\ &= max_a \mathbb{E}_{\pi^*}[G_t|S_t = s, A_t = a] \\ &= max_a \mathbb{E}_{\pi^*}[R_{t+1} + \gamma G_{t+1}|S_t = s, A_t = a] \\ &= max_a \mathbb{E}[R_{t+1} + \gamma v_+(S_{t+1})|S_t = s, A_t = a] \\ &= max_a \sum_{s',r} p(s',r|s,a)[r + \gamma v_*(s')]. \end{aligned} \tag{2.12}$$

The Bellman optimality equation for q_* is:

$$\begin{aligned} q_*(s,a) &= \mathbb{E}[R_{t+1} + \gamma max_{a'} q_*(s_{t+1,a})|S_t = s, A_t = a] \\ &= \sum_{s',r} p(s',r|s,a)[r + \gamma max_{a'} q_*(s',a')] \end{aligned} \tag{2.13}$$

The Bellman optimality equation is actually a system of equations, one for each state. If the dynamics p are known, then in principle one can solve this system for v_* or q_*.

Once one has v_*, it is relatively easy to determine an optimal policy: for each state s, there will be one or more actions at which the maximum is obtained in the Bellman optimality equation. Any policy that assigns nonzero probability only to these actions is an optimal policy; any policy that is greedy with respect to the optimal evaluation function v_* is an optimal policy.

Having q_* makes choosing optimal actions even easier. With q_*, the agent does not ever have to do a one-step-ahead search: for any state s, it can simply find any action that maximizes $q_*(s, a)$.

Explicitly solving the Bellman optimality equation is rarely directly useful. This solution relies on at least three assumptions that are rarely true in practice:

1) we accurately know the dynamics of the environment:

2) we have enough computational resources to complete the computation of the solution;

3) the Markov property.

It is possible to categorize reinforcement learning methods in tabular solution methods and approximate solution methods, but one typically has to settle for approximate solutions.

Since having a model of the environment is not mandatory, it is possible to make another distinction in reinforcement learning: model-based methods and model-free methods. By model we intend a representation of how the world changes in response to the agent's actions.

Model-free algorithms (as opposed to model-based ones) are methods which do not use the transition probability distribution and the reward function associated with the Markov decision process, which represents the problem to be solved. A model-free RL algorithm can be thought of as an explicit trial-and-error algorithm: the agent only experiences what happens for the actions it tries. In a model-free setting it can happen that the value function and/or the policy are known.

Model-based settings are the ones in which the model is explicit, while the policy and/or value function may or may not be known. Clearly, model-based methods rely on planning as their primary component, while model-free methods primarily rely on learning.

Model-free methods are fundamental and widely used, since it is not always possible or it is too difficult to construct a sufficiently accurate environmental model.

2.3.2. Reinforcement Learning methods

In reinforcement learning it is possible to distinguish between tabular solution methods and approximate solution methods. We will describe them briefly, explaining when it is most appropriate to use them.

Tabular solution methods

Tabular solution methods are the ones in which the state and action spaces are small enough for the approximate value functions to be represented as arrays or tables. In this case, the methods can often find exactly the optimal value function and the optimal policy.

There are three fundamental classes of methods for solving finite Markov decision problems: dynamic programming, Monte Carlo methods, and temporal-difference learning. It is also possible to combine these methods in order to obtain the best features of each of them.

- **Dynamic programming:** the term dynamic programming refers to a collection of algorithms that can be used to compute optimal policies given a perfect model of the environment as a Markov decision process. The key idea of DP is the use of value functions to organize and structure the search for good policies. Classical DP methods operate in sweeps through the state set, performing an expected update operation on each state. Each such operation updates the value of one state based on the values of all possible successor states and their probabilities of occurring; expected updates are closely related to Bellman equations. When the updates no longer result in any changes in value, convergence has occurred to values that satisfy the corresponding Bellman equation. Just as there are four primary value functions (v_π, v_*, q_π, q_*) there are four corresponding Bellman equations and four corresponding expected updates.

- **Monte Carlo methods:** it is not assumed the complete knowledge of the environment, but experience, that is sample sequences of states, actions, and rewards from actual or simulated interaction with the environment. Learning from actual experience is striking because it does not require prior knowledge of the environment's dynamics, but also learning from simulated experience is quite powerful. Although a model is required, the model only generate sample transitions, not the complete probability distributions of all possible transitions that is required for DP.

  Monte Carlo methods are ways of solving the reinforcement learning problem based on averaging sample returns. To ensure that well-defined returns are available, these methods are usually applied on episodic tasks. Only on the completion of an episode values are estimated and policies changed. Monte Carlo methods can thus be incremental in an episode-by-episode sense, but not in a step-by-step sense.

  This formulation gives at least three kinds of advantages over DP methods. First, they can be used to learn optimal behavior directly from interaction with the environment, with no model of the environment's dynamics. Second, they can be used with simulation or sample models. Third, it is easy and efficient to focus Monte Carlo methods on a small subset of the states; a region of special interest can be accurately evaluated without the expense of evaluating the rest of the state set.

- **Temporal-difference learning:** it is a combination of Monte Carlo ideas and dynamic programming ideas. Like Monte Carlo methods, TD methods can learn directly from raw experience without a model of the environment's dynamics. Like DP, TD methods update estimates based in part on other learned estimates, without waiting for a final outcome. This is the idea that approximate policy and value functions should interact in such a way that they both move toward their optimal values. It can be identified as a central and novel concept to reinforcement learning.

  These methods are the most widely used nowadays in reinforcement learning. This is due to their simplicity: they can be applied online, with a minimal amount of computation, to experience generated from interaction with an environment; they can be expressed completely by single equations that can be implemented with small computer programs. Moreover, TD methods can be also considered outside the context of reinforcement learning: they are general methods for learning to make long-term predictions about dynamical systems.

Approximate solution methods

Approximate solution methods are extension of tabular methods to problems with arbitrarily large state spaces. In many of the tasks to which apply reinforcement learning, the state space is combinatorial and enormous. In such cases, it is not possible to find an optimal policy or the optimal value function, even in the limit of infinite time and data; the goal instead is to find a good approximate solution using limited computational resources.

The problem with large state spaces is not just the memory needed for large tables, but the time and data needed to fill them accurately. In many of the target tasks, almost every state encountered will never have been seen before. To make sensible decisions in such states, it is necessary to generalize from previous encounters with different states that are in some sense similar to the current one: the key concept is generalization.

It is needed to combine reinforcement learning methods with existing generalization methods. The kind of generalization required is often called function approximation, because it takes examples from a desired function and attempts to generalize from them to construct an approximation of the entire function; it is an instance of supervised learning.

There are several cases that can be considered: the one of on-policy training, in which the policy is given and only its value function is approximated; the control case, in which an approximation to the optimal policy is found; the off-policy learning with function approximation and the policy gradient methods, which approximate the optimal policy directly without an approximate value function.

- **On-policy prediction with approximation:** it is an example in function approximation in reinforcement learning in estimating the state-value function from on-policy data. This means approximating v_π from experience generated using a known policy π. The novelty is that the approximate value function is represented not as a table but as a parametrized function form with weight vector $w \in \mathbb{R}^d$. Typically, the number of weights is much less than the number of states, and changing one weight changes the estimated value of many states. Consequently, when a single state is updated, the change generalizes from that state to affect the values of many other states. Such generalization makes the learning potentially more powerful, but also potentially more difficult to manage and understand.

- **On-policy control with approximation:** the aim is to solve the control problem, that is the problem of finding an optimal policy, with the parametric approximation of the action-value function $q(s, a)$.

- **Off-policy methods with approximation:** in off-policy learning we seek to learn a value function for a target policy π, given data due to a different behavior policy b. In the prediction case, both policies are static and given, and we seek to learn either state values $\hat{v} \approx v_\pi$ or action values $\hat{q} \approx q_\pi$. In the control case, action values are learned, and both policies typically change during learning - π being the greedy policy with respect to $\hat{q}$, and b being something more exploratory.

  One reason to seek off-policy algorithms is to give flexibility in dealing with the trade-off between exploration and exploitation. Another is to free behavior from learning, and avoid the tyranny of the target policy. The whole area of off-policy learning is relatively new and unsettled.

- **Policy gradient methods:** these methods learn a parameterized policy that can select actions without consulting a value function. A value function may still be used to learn the policy parameter, but it is not required for action selection. These methods learn the policy parameter based on the gradient of some scalar performance measure with respect to the policy parameter. They seek to maximize the performance, so their updates approximate gradient ascent.

All the methods that follow this general schema are called policy gradient methods, while methods that learn approximations to both policy and value functions are often called actor-critic methods, where actor is a reference to the learned policy, and critic refers to the learned value function.

Methods that learn and store a policy parameter have many advantages: they can learn specific probabilities for taking the actions; they can learn appropriate levels of exploration and approach deterministic policies asymptotically; they can naturally handle continuous action spaces.

2.4. Dynamic Programming in Reinforcement Learning

The following section is a focus on dynamic programming techniques in reinforcement learning: being a first reinforcement learning approach to anomaly detection in temporal graphs, we have decided to focus on DP in order to understand whether this approach gives meaningful results or not.

As introduced before, dynamic programming algorithms can be used to compute optimal policies under the condition of a perfect model of the environment as a Markov Decision process. Classical DP algorithms are of limited utility in reinforcement learning both because of their assumption of a perfect model and of their great computational expense. They still are important theoretically, providing a basis for more evolved methods.

The starting assumption is that the environment is a finite MDP, that is we assume that its states, action, and reward sets, S, A, and R, are finite, and that its dynamics are given by a set of probabilities $p(s', r|s, a)$ for all $s, s' \in S$, $a \in A$, and $r \in R$.

The central idea is that of using value functions to set up the search for good policies. As discussed before, it is possible to easily find the optimal policies once the optimal value functions, which satisfy the Bellman optimality equation, have been found:

$$\begin{aligned} v_*(s) &= \max_a \mathbb{E}[R_{t+1} + \gamma v_*(S_{t+1})|S_t = s, A_t = a] \\ &= \max_a \sum_{s',r} p(s', r|s, a)[r + \gamma v_*(s')], or \end{aligned} \tag{2.14}$$

$$\begin{aligned} q_*(s, a) &= \mathbb{E}[R_{t+1} + \gamma \max_{a'} q_*(s_{t+1,a})|S_t = s, A_t = a] \\ &= \sum_{s',r} p(s', r|s, a)[r + \gamma \max_{a'} q_*(s', a')] \end{aligned} \tag{2.15}$$

for all $s, s' \in S$, $a \in A$.

DP algorithms are obtained by turning Bellman equations into assignments, that is into update rules for improving the approximation of the value functions.

2.4.1. Policy Evaluation or Prediction

By policy evaluation or prediction problem is intended the issue of how to compute the state-value function v_π for an arbitrary policy π. We already know that for all $s \in S$:

$$\begin{aligned} v_\pi(s) &= \mathbb{E}_\pi[G_t|S_t = s] \\ &= \mathbb{E}_\pi[R_{t+1} + \gamma G_{t+1}|S_t = s] \\ &= \mathbb{E}_\pi[R_{t+1} + \gamma v_\pi(S_{t+1})|S_t = s] \\ &= \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a)[r + \gamma v_\pi(s')] \end{aligned} \tag{2.16}$$

The existence and uniqueness of v_π are guaranteed if either $\gamma < 1$ or if termination is guaranteed from all states under the policy π.

If the dynamics of the environment are completely known, then we have a system of $|S|$ linear equations, where the $|S|$ unknowns are the $v_\pi(s)$ for $s \in S$. It is possible to directly solve this system, but since it is computationally expensive it is better to use iterative solution methods.

Consider a sequence of approximate value functions v_0, v_1, v_2, ..., each mapping S to $\mathbb{R}$. The initial approximation v_0 is chosen in an arbitrary way, except that the terminal state, if present, must be given value 0. Each successive approximation is obtained using the Bellman equation of v_π as update rule:

$$\begin{aligned} v_{k+1}(s) &= \mathbb{E}_\pi[R_{t+1} + \gamma v_k(S_{t+1}) | S_t = s] \\ &= \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a)[r + \gamma v_k(s')] \end{aligned} \tag{2.17}$$

for all $s \in S$.

Clearly, when the updates no longer result in any changes in value it means that convergence has happened to values that satisfy the Bellman equation. In fact, $v_k = v_\pi$ is a fixed point for this update rule. The sequence $\{v_k\}$ can be shown to converge to v_π as $k \to \infty$ under the same conditions that guarantee the existence of v_π. This algorithm is called iterative policy evaluation.

The operation performed above is called expected update: to obtain v_{k+1} from v_k, for each state it replaces the old value of s with a new value obtained from the old values of the successor states of s, and the expected immediate rewards, along all the one-step transitions possible under the policy being evaluated.

There are several different kinds of expected updates, depending on whether the state or a state-action pair is updated, and also according to the way the estimated values of the successor states are combined. These updates are called expected because they are based on an expectation over all possible next states rather than on a sample next state.

2.4.2. Policy Improvement

Once we have determined the value function v_π for an arbitrary policy π, the next step is to understand whether for some state s it is better to change the policy to deterministically choose an action $a \neq \pi(s)$. One way to answer this question is to consider the action a in the state s and then following the existing policy π, that is:

$$\begin{aligned} q_\pi(s, a) &= \mathbb{E}[R_{t+1} + \gamma v_\pi(S_{t+1}) | S_t = s, A_t = a] \\ &= \sum_{s',r} p(s', r|s, a)[r + \gamma v_\pi(s')] \end{aligned} \tag{2.18}$$

We need to evaluate if this quantity is greater or less than $v_\pi(s)$: if it is greater, it means that it is better to select a when in s and then follow the policy π rather than follow π all the time. As a consequence, the modified policy in that selects a every time s is encountered is better than π.

This is a special case of a general result called the policy improvement theorem:

[Policy improvement theorem] Let π and π' be any pair of deterministic policies such that for all $s \in S$ it holds that $q_\pi(s, \pi'(s)) \geq v_\pi(s)$.

Then the policy π' must be as good as, or better than, π, that is it must obtain greater or equal expected return from all states $s \in S$: $v'_\pi(s) \geq v_\pi(s)$.

Moreover, if there is a strict inequality $q_\pi(s, \pi'(s)) > v_\pi(s)$ at any state, then there must be a strict inequality also in $v'_\pi(s) > v_\pi(s)$ at that state. Thus, if $q_\pi(s, a) > v_\pi(s)$, then the changed policy is indeed better than π.

We have seen how, given a policy and its value function, we can evaluate the consequences of a change in the policy at a single state to a particular action. The natural extension is to consider the changes at all states and to all possible actions, selecting at each state the actions that seems best according to $q_\pi(s,a)$. The new considered policy π' is given by:

$$\begin{aligned}\pi'(s) &=_a q_\pi(s,a) \\ &=_a \mathbb{E}[R_{t+1} + \gamma v_\pi(S_{t+1})|S_t = s, A_t = a] \\ &=_a \sum_{s',r} p(s',r|s,a)[r + \gamma v_\pi(s')].\end{aligned} \tag{2.19}$$

The greedy policy π' takes the action that looks best in the short term, that is one step ahead, according to v_π. By definition, π' satisfies the conditions of the policy improvement theorem, so it is as good as, or better than, the original policy. This process is called policy improvement.

Suppose that the new greedy policy π' is as good as, but not better than, the old policy π. Then $v_\pi = v'_\pi$, and from (2.19) it is possible to derive that for all $s \in S$:

$$\begin{aligned}v'_\pi(s) &= \max_a \mathbb{E}[R_{t+1} + \gamma v'_\pi(S_{t+1})|S_t = s, A_t = a] \\ &= \max_a \sum_{s',r} p(s',r|s,a)[r + \gamma v'_\pi(s')], or\end{aligned} \tag{2.20}$$

This is exactly the Bellman optimality equation (2.14), therefore v'_π must be v_*, and both π and π' must be optimal policies.

The policies considered up to now are deterministic, but these ideas can be extended to stochastic policies, that specify probabilities $\pi(a|s)$ for taking each action a in each state s.

2.4.3. Policy Iteration

Once a policy π has been improved using v_π, and a better policy π' has been obtained, it is possible to compute v'_π and improve it again to obtain an even better policy π''. In this way we get a sequence of monotonically improving policies and value functions:

$$\pi_0 \xrightarrow{E} v_{\pi_0} \xrightarrow{I} \pi_1 \xrightarrow{E} v_{\pi_1} \xrightarrow{I} \pi_2 \xrightarrow{E} \dots \xrightarrow{I} \pi_* \xrightarrow{E} v_*,$$

where $\xrightarrow{E}$ denotes a policy evaluation and $\xrightarrow{I}$ denotes a policy improvement.

Since a finite MDP has only a finite number of policies, the process must converge in a finite number of iterations to an optimal policy and an optimal value function. This procedure of finding an optimal policy is called policy iteration.

One drawback of policy iteration is that each iteration involves policy evaluation, which may itself be an iterative computation: it is possible to find a way to avoid this issue.

2.4.4. Value Iteration

In order to improve policy iteration, one must consider that the policy evaluation step of policy iteration can be truncated in several ways without losing convergence. One particular case is when policy evaluation is stopped after just one update of each state (that is one sweep): the value iteration algorithm.

This algorithm can be written as a simple update operation that combines the policy improvement and truncated policy evaluation step:

$$\begin{aligned}v_{k+1} &= \max_a \mathbb{E}[R_{t+1} + \gamma v_k(S_{t+1}))|S_t = s, A_t = a] \\ &= \max_a \sum_{s',r} p(s',r|s,a)[r + \gamma v_k(s')]\end{aligned} \tag{2.21}$$

for all $s \in S$. For an arbitrary v_0, it is possible to prove that the sequence $\{v_k\}$ converges to v_* under the same conditions that guarantee the existence of v_*.

In order to understand the key idea of value iteration, we can consider the Bellman optimality equation (2.14). Value iteration, in fact, is obtained by turning the Bellman optimality equation into an update rule.

Formally, value iteration requires an infinite number of iterations to converge: in practice, we stop when the value function changes only by a small amount in a sweep. A faster convergence is often achieved by interposing multiple policy evaluation sweeps between each policy improvement sweep. However, in general, the whole class of truncated policy iteration algorithms can be considered as sequences of sweeps, some performing policy evaluation updates and others performing value iteration updates.

2.4.5. Asynchronous Dynamic Programming and Generalized Policy Iteration

One drawback of the methods discussed before, is that they involve operations that need to be performed over the entire state set of the MDP: if the state set is very large, it can become too much expensive. Asynchronous DP algorithms are in-place iterative algorithms that are not organized in terms of systematic sweeps of the state set: they update the values of the state in any order, using the values of others states that are available. In order to converge correctly, it is still necessary to update the values of all the states: what changes is the great flexibility in selecting states to update.

This technique does not necessarily result in less computation, it simply means that the algorithm does not get stuck into a long sweep before it can improve the policy. It is possible to organize the order of the updates in order to take advantages from this strategy, for example trying to order the updates to let value information propagate from state to state in an efficient way. It is also possible to combine asynchronous algorithms with real time interaction: the agent's experience can be used to determine the states to update, and, at the same time, the latest value and policy information from the algorithm can guide the agent's decision making.

Policy iteration consists of two simultaneous interacting processes, policy evaluation and policy improvement. In policy iteration, these two processes alternate, one beginning when the other finishes, but this is not strictly necessary as we have seen in value iteration and asynchronous algorithms. The term generalized policy iteration (GPI) refers to the idea of letting policy evaluation and policy improvement processes interact, independently from the details of the two processes. When both the processes stabilize, in the sense that they do not change, then the value function and policy must be optimal. The value function stabilizes only when it is consistent with the current policy, and the policy stabilizes only when it is greedy with respect to the current value function.

The two processes in GPI are both competing and cooperating. They compete because making the policy greedy with respect to value function typically makes the value function incorrect for the changed policy, and making the value function consistent with the policy typically causes the policy no longer to be greedy. However, in the long turn these processes interact to find a single common solution: the optimal value function and an optimal policy.

In conclusion, considering the efficiency of dynamic programming, we can say that DP may not be practical for solving very large problems, but compared with other methods for solving MDPs it is quite efficient. The worst case time DP methods take to find an optimal policy is polynomial in the number of states and actions. In this sense, DP is exponentially faster than any direct search in policy space could be, because direct search needs to examine each possible policy to guarantee the finding of the optimal one.

Linear programming methods can also be used to solve MDPs, and it happens that their performance is better than DP, but the problem is that linear programming becomes impractical at a much smaller number of states than do DP methods. On the other hand, sometimes also DP methods are of limited applicability due to the curse of dimensionality, that is the fact that the number of states often grows exponentially with the number of state variables. Anyways, DP methods remain the best option for handling large state spaces compared to direct search and linear programming: today's computers are able to solve MDPs with millions of states. Clearly, in cases of large state spaces, asynchronous methods and GPI perform better than synchronous methods.

2.5. Graph generators

Since the report is focused on the application of reinforcement learning techniques on small graphs without anomalies at first, and since it is very difficult in the real world to find such graphs, we introduce graph generators.

Graphs are ubiquitous in the real world, representing objects and their relationships such as social networks, citation networks, biology networks, traffic networks, etc. Graphs are also known to have complicated structures that contain rich underlying values. Much effort has been made in this area, resulting in a rich literature of related papers and methods to deal with various kinds of graph problems, which can be categorized into two types: 1) predicting and analyzing patterns on given graphs; 2) learning the distributions of given graphs and generating more novel graphs. We are interested in this second problem.

As explained in [8], graph generation entails modeling and generating real-world graphs, and it has applications in several domains, such as understanding interaction dynamics in social networks, link prediction, and anomaly detection. Owing to its many applications, the development of generative models for graphs has a rich history, resulting in famous models such as random graphs, small-world models, stochastic block models, and Bayesian network models, which generate graphs based on apriori structural assumptions.

2.5.1. Erdős–Rényi Model

The Erdős–Rényi random graph model, that is the mother of all graph models as explained in [9], was introduced in 1959 by the Hungarian mathematicians Paul Erdős and Alfréd Rényi.

The method starts with N vertices, and randomly links two nodes with a probability of p. Such graphs with N nodes and edge probability p form a set of graphs $G_{N,p}$.

A graph in $G_{N,p}$ has on average $p\binom{n}{2}$ edges. The distribution of the degree of any particular vertex is binomial:

$$P(deg(v) = k) = \binom{N-1}{k} p^k (1-p)^{N-1-k}$$

Since

$$P(deg(v) = k) \to \frac{(Np)^k e^{-Np}}{k!},$$

this distribution is Poisson for large N and Np constant.

In a 1960 paper, Erdős and Rényi described the behavior of $G_{N,p}$ very precisely for various values of p. Their results included that:

- If $Np < 1$, then a graph in $G_{N,p}$ will almost surely have no connected components of size larger than $O(log(N))$.
- If $Np = 1$, then a graph in $G_{N,p}$ will almost surely have a largest component whose size is of order $N^{2/3}$.
- If $Np \to c > 1$, where c is a constant, then a graph in $G_{N,p}$ will almost surely have a unique giant component containing a positive fraction of the vertices. No other component will contain more than $O(log(N))$ vertices.
- If $p < \frac{(1-\epsilon)lnN}{N}$, then a graph in $G_{N,p}$ will almost surely contain isolated vertices, and thus be disconnected.
- If $p > \frac{(1+\epsilon)lnN}{N}$, then a graph in $G_{N,p}$ will almost surely be connected.

Thus $\frac{lnN}{N}$ is a sharp threshold for the connectedness of $G_{N,p}$.

Further properties of the graph can be described almost precisely as N tends to infinity.

Several other random networks have been proposed that change some of the properties of the initial Erdős and Rényi random networks. The most obvious change to $G_{N,p}$ is the change of the degree distribution from a Poisson to a power law.

2.5.2. Stochastic Block Model

The stochastic block model is a generative model for random graphs. This model tends to produce graphs containing communities, subsets characterized by being connected with one another with particular edge densities: for example, edges may be more common within communities than between communities. Its mathematical formulation has been firstly introduced in 1983 in the field of social network by Holland. The stochastic block model is important in statistics, machine learning, and network science, where it serves as a useful benchmark for the task of recovering community structure in graph data.

The stochastic block model takes the following parameters: the number N of vertices; a partition of the vertex set $\{1, \ldots, N\}$ into disjoint subsets $\{C_1, \ldots, C_r\}$ called communities; a symmetric $r \times r$ matrix P of edge probabilities.

The edge set is then sampled at random as follows: any two vertices $u \in C_i$ and $v \in C_j$ are connected by an edge with probability P_{ij}. If the probability matrix is a constant, in the sense that $P_{ij} = p$ for all i and j, then the result is the Erdős–Rényi model $G_{N,p}$. This case is degenerate since the partition into communities becomes irrelevant, but it illustrates a close relationship to the Erdős–Rényi model.

The planted partition model is the special case that the values of the probability matrix P are a constant p on the diagonal and another constant q off the diagonal. Thus two vertices within the same community share an edge with probability p, while two vertices in different communities share an edge with probability q. Sometimes it is this restricted model that is called the stochastic block model. The case where $p > q$ is called an assortative model, while the case $p < q$ is called disassortative.

For example, we can think of a graph produced by the SBM with the following features: $V = \{1, 2, 3, \ldots, 90\}$ is the vertex set, the disjoint communities are $C = \{C_1, C_2, C_3\}$, where C_1 has 25 nodes ($C_1 = \{1, 2, 3, \ldots, 25\}$), C_2 has 30 nodes ($C_2 = \{26, 27, 28, \ldots, 55\}$, and C_3 has 35 nodes ($C_3 = \{56, 57, 58, \ldots, 90\}$. We could assume that the probability of having an edge between nodes belonging to the same community is 0.8 for the three communities, while the probability of an edge between nodes in two different communities is equal to 0.05. In this case the probability matrix P, assuming that the rows and columns are the communities, has the form:

$$P = \begin{pmatrix} 0.8 & 0.05 & 0.05 \\ 0.05 & 0.8 & 0.05 \\ 0.05 & 0.05 & 0.8 \end{pmatrix}$$

An example of graph with these characteristics is:

The adjacency matrix A of the graph, in which on the row and the columns there are the nodes, has the entry a_{ij} equal to 1 if there is an edge between node i and j, otherwise it is equal to 0. It is a representation of the existing relations in the graph and it has the following form:

The matrix is symmetric since the graph is undirected. It is possible to notice how the matrix is almost perfectly divided into three blocks, that correspond to the three communities: the only elements that lie outside the blocks are the ones representing the edges between nodes belonging to different communities.

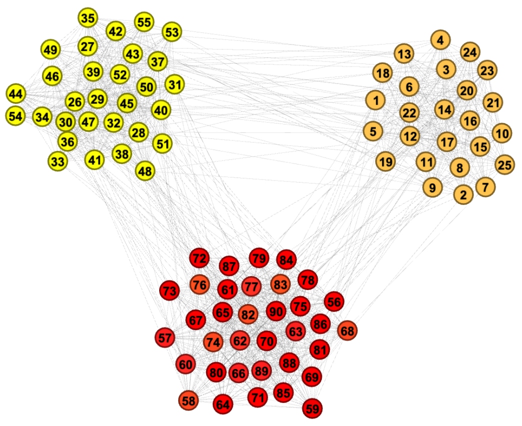


Figure 2.2: Example of graph generated by the Stochastic Block Model.

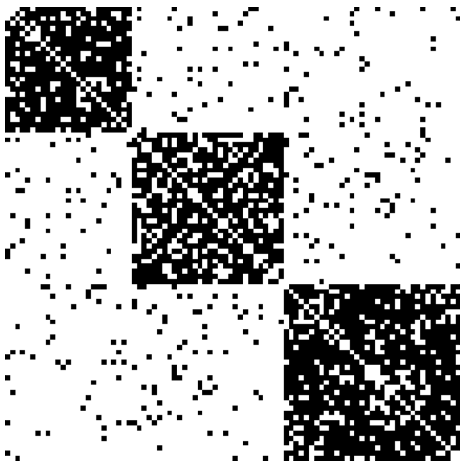


Figure 2.3: Adjacency matrix related to a graph generated by the Stochastic Block Mode.

3 Modelisation of Anomaly Detection in Dynamic Graphs for Reinforcement Learning

This chapter is dedicated to the formalization and application of dynamic programming methods to detect anomalies in temporal graphs. In particular, the type of anomaly to detect is the one on edges. Our aim is to explain how, given an undirected temporal graph as input, we model it as a Markov Decision Process and how we can use dynamic programming methods to train an agent to detect anomalies. Another fundamental step consists in the formal definition of the anomalies: we need to define what are anomalies in this case, how it is possible to identify them and which is the ranking model to adopt. The chapter starts with a recall of the theoretical concepts that will be exploited, and then proceeds with the formalization steps mentioned before. Besides, the last section provides an overview of the possible practical uses of the proposed detector.

3.1. Theoretical concepts

Before formalizing the problem, let us provide more details on the theoretical underlying concepts, briefly introduced in the previous chapter, that are needed in this section.

As defined in (2.2), a graph series $\mathbb{G}$ (or dynamic graph) is an ordered set of graphs with a fixed number of time steps. Formally, $\mathbb{G} = \{G_t\}_{t=1}^{T}$, where T is the total number of graphs, and $G_t = (V, E_t \subseteq (V \times V))$, with V the vertex set, that is considered as fixed, and E_t the edge set at time step t. Graph series where $t \to \infty$ are called graph streams.

In our specific case, the type of anomaly to detect is the one on arcs, intended as three different situations:

1) it can happen that the action expected is the addition of a certain arc, instead the performed action is the addition of another arc, the removal of an arc or everything stays the same. We mark this setting as anomalous, but it is necessary to understand why both the first arc is not present and another arc is added/removed, or why nothing happens;

2) we mark as anomalous the situation in which the agent foresees the removal of a certain edge, while in the real graph another arc is added/removed or nothing happens. As before, we need to understand why the arc has not been removed and why another arc has been added/removed, or why nothing happens;

3) the last case is when according to the agent the situation should remain unchanged, while actually an arc is added or removed: we mark this addition/removal as anomalous.

The common element of these three cases is that we mark as anomalous every situation in which there is no coincidence between what the agent expects and what actually happens.

Recalling definition (2.2.1.2), given $\mathbb{G}$, the total edge set $E = \bigcup_{t=1}^{T} E_t$, and a specified scoring function $f : E \to \mathbb{R}$, the set of anomalous edges $E' \subseteq E$ is an edge set such that $\forall$ e' $\in E'$, $|f(e') - \hat{f}| > c_0$, where $\hat{f}$ is a summary statistic of the scores f(e), $\forall$ e $\in E$, and c_0 a fixed threshold.

In order to detect anomalies, we would like to apply dynamic programming techniques: the topic has been widely treated in section 2.4. One requisite to apply DP methods is a known environment modelled as Markov Decision Process: it is a tool for modeling sequential decision-making problems where a decision maker interacts with a system in a sequential fashion. A Markov Decision Process is a 4-tuple (S, A, p, R), where:

- S is the set of states called the state space;
- A is the set of actions called the action space (alternatively, $A(s)$ is the set of actions available from state s);
- $p(s', r|s, a) = P\{S_t = s', R_t = r|S_{t-1} = s, A_{t-1} = a\}$ is the probability that action a in state s at time t will lead to state s' and reward r at time $t + 1$;
- $R(s, s', a)$ is the immediate reward (or expected immediate reward) received after transitioning from state s to state s', due to action a.

The purpose of reinforcement learning is let the agent learn an optimal policy that maximizes the reward function. A basic reinforcement learning agent interacts with its environment in discrete time steps: at each time t, the agent receives the current state s_t and reward r_t. It then chooses an action a_t from the set of available actions, which is subsequently sent to the environment. The environment moves to a new state s_{t+1} and the reward r_{t+1} associated with the transition (s_t, a_t, s_{t+1}) is determined. The goal of a reinforcement learning agent is to learn a policy: $\pi : A \times S \to [0, 1]$, $\pi(a, s) = \Pr(a_t = a \mid s_t = s)$ which maximizes the expected cumulative reward. Formulating the problem as an MDP assumes the agent directly observes the current environmental state; in this case the problem is said to have full observability.

Dynamic programming methods reach this goal by the use of value functions to organize and structure the search for good policies. As explained in section 2.4 and in [10], the two most famous and used methods are policy iteration and value iteration.

Policy iteration works in this way: first, we fix an arbitrary initial policy π_0. At iteration $k > 0$, we compute the action-value function q_{π_k} (defined in 2.3.1) underlying π_k: this is called the policy evaluation step. Next, given q_{π_k}, we define π_{k+1} as a policy that is greedy with respect to q_{π_k}: this is called the policy improvement step. We now make an example of policy iteration, taken from [2].

The example is related to car rental: let us suppose that Jack manages two locations for a car rental company. Each day a certain number of customers go to the locations to rent cars: if Jack has an available car, then he earns \$10, if he doesn't have disposable cars at that location the affair is lost. A car becomes available for renting the day after it is returned. In order to be sure to have enough available cars, Jack can move them between the two locations during the night, paying \$2 per car. We assume that:

- the number of cars requested and returned at each location are Poisson random variables, that is the probability that the number is n is $\frac{\lambda^n}{n!}e^{-n}$, where λ is the expected number;
- at the first location λ is 3 for rental request and 3 for returns,
- at the second location λ is 4 for rental request and 2 for returns;
- there can be at most 20 cars at each location (additional cars will not be considered;
- a maximum of 5 cars can be moved from one location to the other.

We assume the discount rate to be $\gamma = 0.9$ and formulate the problem as a continuing finite MPD, where the time steps are the days, the state is the number of cars at each location at the end of the day, and the actions are the net number of cars moved during the night.

The picture in the following page shows the sequence of policies found by policy iteration starting from the policy that never moves any cars. In particular, the first five diagrams show, for each number of cars at each location at the end of the day, the number of cars to be moved from the first location to the second (negative numbers indicate transfers from the second location to the first). Each successive policy is a strict improvement over the previous one, as assured by the policy improvement theorem, and the last policy is optimal. Policy iteration converges in surprisingly few iterations as it often happens, in this case just 4.

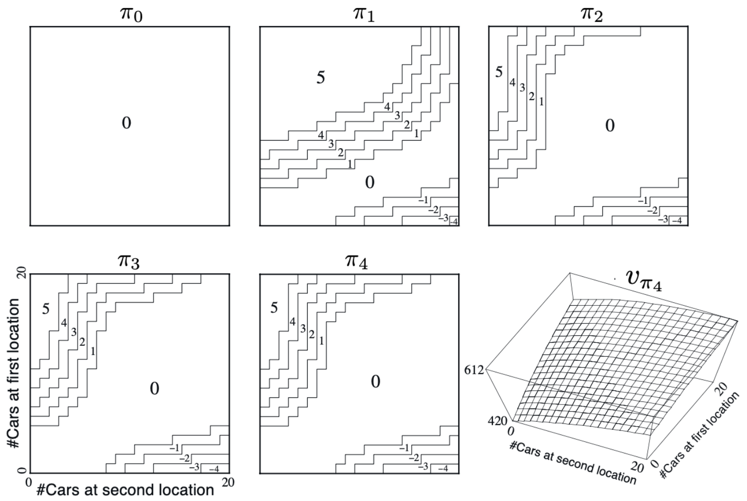


Figure 3.1: The sequence of policies found by policy iteration on Jack's car rental problem (from [2]), and the final state-value function.

Value iteration can be written as a simple update operation that combines the policy improvement step and the truncated policy evaluation step. In particular, policy evaluation is stopped after just one update of each state: for this reason, the computational cost of a single step in policy iteration is much higher than that of one update in value iteration. However, after k iterations, policy iteration gives a policy not worse than the policy that is greedy with respect to the value function computed using k iterations of value iteration (if the two procedures are started with the same initial value function).

The following example of value iteration is taken from [2], and it is entitled "gambler's problem". A gambler has the opportunity to bet on the results of a sequence of coin flips: if the coin comes up heads, he wins as many dollars as he has put on that flip; if it is tails, he loses his stake. The game ends when the gambler wins by reaching his goal of \$100, or loses by running out of money. On each flip, the gambler must decide what portion of his capital to stake, in integer numbers of dollars.

This problem can be formulated as an undiscounted, episodic, finite MDP. The state is the gambler's capital, $s \in \{1, 2, ..., 99\}$, and the actions are stakes, $a \in \{0, 1, ..., \min(s, 100 - s)\}$. The reward is zero on all the transitions, except those on which the gambler reaches his goal, then it is +1. The state-value function gives the probability of winning from each state, and a policy is a mapping from levels of capital to stakes: the optimal policy maximizes the probability of reaching the goal. Let p_h denote the probability of the coin coming up heads: if it is known, then the entire problem is known and it can be solved, for instance, by value iteration.

The policy found is optimal, but not unique. In fact, there is a whole family of optimal policies, all corresponding to ties for the action selection with respect to the optimal value function.

The main limit of dynamic programming in reinforcement learning is computational: apart from the simplest cases when the MDP has few states and actions, dynamic programming becomes infeasible. As mentioned before, the problem is the curse of dimensionality, that is the fact that the number of states often grows exponentially with the number of state variables. Moreover, having a perfect model of the environment is not always possible in the real world. However, DP provides an essential foundation for the understanding and developments of successive reinforcement learning methods.

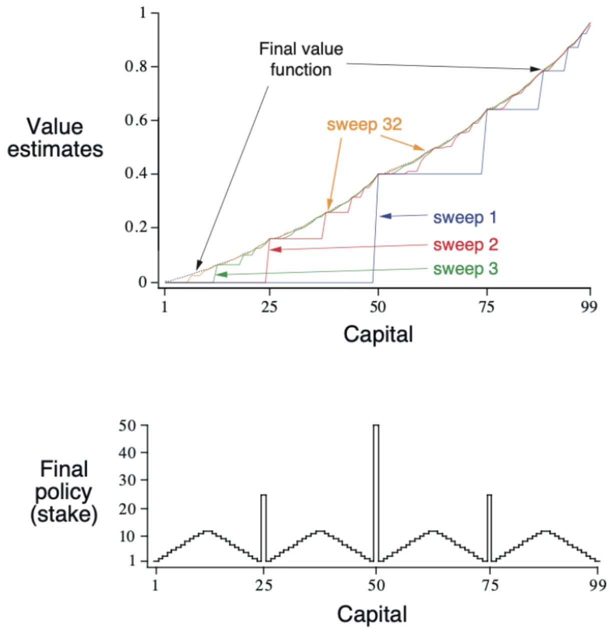


Figure 3.2: The figure (provided in [2]) shows the change in the value function over successive sweeps of value iteration, and the final policy found, for the case of $p_h = 0.4$.

3.2. Formalization of the problem

This section contains the modeling and formalization of the anomaly detection problem in our particular setting. First, the attention is focused on how to model the undirected temporal graph in input into a Markov Decision Process, how to understand the dynamics of the environment and how to set the rewards. Once this is done, we need to formalize what we mean by edge anomalies, how the anomalies are identified, and how to rank them. The last step concerns the application of dynamic programming methods: we will do a practical example in order to understand better.

3.2.1. Graph modeling

The input of our problem is an undirected temporal graph, but in order to apply dynamic programming methods we need to have the environment modeled as Markov Decision Process. One important prerequisite of the input graph is to be structured in the following way: each time step t has to corresponds to the addition of an arc, the removal of an arc or the graph does not change. In the cases in which we do not have this setting, it is necessary to perform some preliminary operations in order to reach it. Once this condition is achieved, starting from the dynamic graph we have to define the 4-tuple (S, A, p, R) mentioned above.

The set of states S is made by all the possible configurations of the graph. We assume for simplicity that the set of nodes V is fixed. Since the anomalies to detect are on the arcs, the set of states S contains all the possible combinations of the arcs between the existing nodes: each combination constitutes a state s. Assuming that the number of nodes is n, that is $|V| = n$, the total number of arcs is equal to $|E| = \frac{n(n-1)}{2}$ (the quantity $n(n-1)$ is divided by 2 since the considered graph is undirected). The cardinality of S is given by all the possible subsets of E, that is

$$|S| = 2^{|E|} = 2^{\frac{n(n-1)}{2}} \sim 2^{\frac{n^2}{2}}. \tag{3.1}$$

It is easy to understand how fast the number of states grows with respect to the number of nodes, and how the dynamic programming methods can become computationally infeasible.

For what concerns the set of actions A, the possible actions are 3: add an arc, remove an arc or do nothing. So, if the the cardinality of the edge set is $|E| = e$, then the possible actions are

$$|A| = e + e + 1 = 2e + 1 \sim e \tag{3.2}$$

The first e actions are the ones relative to the addition of the arcs (one for each arc), the second term is relative to the removal of each arc, and the 1 is the action 'do nothing'. Clearly, not all the actions can be performed in any state: for example, if an arc is not present, it is not possible to remove it. If the agent chooses to perform an action that is not allowed, the graph configuration remains the same and the reward is zero. This setting aims to be as general as possible: then, depending on the cases, it can be modified.

The function p, that defines the dynamics of the environment, depends on the specific problem. In our case it describes the probability that, being in a certain configuration s of the graph at time t and performing the action a of adding/removing an arc or doing nothing, we will get at time $t + 1$ a certain new configuration s' and a reward r. We need to have a perfect knowledge of the dynamics that regulate the graph: this means that another fundamental characteristic of the graph we have in input is that it is without anomalies, and that the sequence is long enough to be able to extrapolate the normal behavior. To make the agent learn what is normal, we define the dynamics of the environment in a way that the only situations that are positively rewarded are the ones present in the clean sequence. In the implementation chapter we will explain how it is actually done, fixing some appropriate values for the rewards. Obviously, reasoning in this way the involved policies are stochastic: it can happen that one state is present several times in the sequence, and that the action performed in the state is not always the same. So, we affirm that more than one action is possible from states that appear more than one time, and the probability of each action of being performed depends on how many times that action is performed in that state considering the whole sequence.

The reward set R is user-defined and problem-dependent, so it is our task to determine it in the best possible way. In fact, it is our way of communicating to the agent what we want to achieve, and if it is not settled appropriately the agent will not be able to learn correctly. The idea is to base the definition of the reward function on the clean sequence that we have: through the sequence, we have to find common patterns and behaviors. In this case it could be a good idea to define the reward set $R = [0, 1]$, giving a reward of 1 when the state transition is present in the clean sequence and 0 otherwise.

In order to understand better, we can make an example. Let us suppose that the graph in the picture is the undirected dynamic graph in input and that our aim is to model it as explained. In this case we represent only the first 5 time steps.

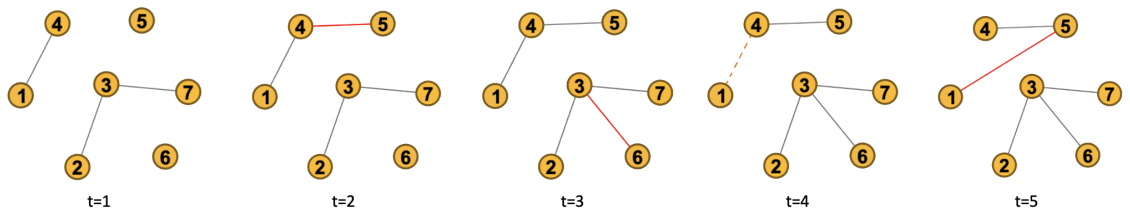


Figure 3.3: Example of a temporal graph captured in 5 subsequent time steps.

The graph is composed by $|V| = 7$ nodes, where $V = \{1, 2, 3, 4, 5, 6, 7\}$, so the total possible number of arcs is $|E| = 21$, the number of states is 2^{21} (given by (3.1)), and the number of actions is 43 (given by (3.2)). As it is possible to notice, it looks as if there are two different communities in the graph, that are $C_1 = \{1, 4, 5\}$ and $C_2 = \{2, 3, 6, 7\}$. The red arcs are the ones that are added between two consecutive steps, while those dashed in orange are the arcs removed.

It is not practical to enumerate all the possible states and actions, but we can make some examples: some possible states are given by the 5 snapshots represented in the picture. For what concerns the actions, we have:

- do nothing;
- add an arc between node 1 and 2;
- remove an arc between node 1 and 2;
- add an arc between node 1 and 3;
- ...

As said, not all the actions are possible in every state: if the agent selects an action that is not allowed (for example it removes an arc that is not present) the situation will remain the same.

Considering the reward function and the dynamic function, following the idea we have already presented, we can give a reward of 1 when the transition is present in the sequence and 0 otherwise. For this reason, as anticipated, the idea is to have a sufficiently long sequence of the temporal graph without anomalies.

Clearly, the assumption of having such clean sequence is quite unrealistic, as it is the assumption of having a perfect model of the environment (the two things coincide in this case). As explained before, the aim of the report is to understand if the reinforcement learning approach is meaningful for the anomaly detection problem on temporal graphs: according to the results obtained in the experimental part in Chapter 4, we will decide how to proceed.

3.2.2. Formalization of the anomaly detection problem

It is necessary to define what we mean by anomalies, how we find them and how to assign an anomaly score to rank them. As said before, the anomaly is on the edges.

First of all we need to understand how to identify the anomalies: the basic concept is that we consider as anomalous every edge that our trained agent does not foresee. This means that, once the detector is ready, it will make forecasts about the behavior of the graph in the future: if the behavior of the agent and the one of the graph coincide, then we will not have anomalies, otherwise we will. By behavior we intend the edge added/removed or the action of doing nothing at each time step: if according to the agent the action at a certain time t is the addition of an arc between two certain nodes i and j, while in the graph there is the addition of an arc between the nodes j and k with $k \neq i$, then the edge $\{j, k\}$ will be marked as anomalous. It is assumed, in fact, that the trained agent has learnt what is normal for the evolution of the graph, so when there is no coincidence between an action that he makes and the actual evolution of the graph, we suppose that there is an anomalous situation.

Clearly there is also the necessity to rank the anomalies, that is to assign them a score in order to understand their gravity. In this case, we have two options that we can follow: binary or non-binary score. The binary solution, that is the simplest one, does not actually allow us to rank the anomalies, since it only assigns 1 to the edge if it is anomalous and 0 otherwise. So, roughly speaking, when there is no coincidence between the action expected by the agent and the one happened in the graph, we mark the situation as anomalous, but it is not possible to say anything about the severity of the anomaly itself.

The second option, instead, aims not only to identify the anomaly, but also to give it a score in order to be able to make a ranking. The principle in identifying the anomaly is the same: when there is no correspondence between what expected by the agent and what actually happens, we have an anomaly. We can now consider the two situations, that is the expected one and the actual one: we have to find a way to compare them in order to evaluate the importance of the anomaly. One possibility could be to use the value function: let us recall that the value function $v_\pi(s)$ is a measure of the long-term desirability of the state after taking into account the states that are likely to follow according to policy π and the rewards available in those states. We seek actions that bring about states of highest value, not highest reward, because these actions obtain the greatest amount of reward in the long run. We need to remember that we have already found the optimal policy π_*, so the value function $v_{\pi^*}(s)$ is already the highest one for each $s \in S$. What we do is compute the value function at the two states s (the expected one) and s' (the actual one), make the subtraction between the value of the expected state and the value of the actual state and take this quantity as anomaly score. What we expect, if the problem is well formalized, is that $v_{\pi^*}(s') \leq v_{\pi^*}(s)$, otherwise the policy π^* would not be optimal. If so, if the difference is performed in the correct order, the anomaly score will always be positive, otherwise the policy is not optimal.

One thing to do is to normalize this quantity: what we want to do is to measure the gravity of the anomaly considering how much the value function is reduced between the foreseen situation and the real one. Without the normalization, it could happen that an anomaly score is much higher than another one, but just because the value function in consideration are higher in magnitude. For example, it may happen that $v_{\pi^*}(s_1) - v_{\pi^*}(s_2) = 10 - 1 = 9$ and $v_{\pi^*}(s_3) - v_{\pi^*}(s_4) = 50 - 25 = 25$. Obviously $25 > 9$, but in the first case the value function $v_{\pi^*}(s_2)$ is $\frac{1}{10}$ of $v_{\pi^*}(s_1)$, while in the other case it is just $\frac{1}{2}$, so the score does not perfectly reflect the importance of the anomaly.

We could define the anomaly score as $1 - \frac{v_{\pi^*}(s')}{v_{\pi^*}(s)}$, that is 1 minus the ratio between the value function of the actual state and the value function of the expected one. Again, if the problem is well formalized, we get that $v_{\pi^*}(s') \leq v_{\pi^*}(s)$, so $0 \leq \frac{v_{\pi^*}(s')}{v_{\pi^*}(s)} \leq 1$ and the anomaly score is between 0 and 1. In this way, we get a score that is normalized and that takes into account the rate of change of the value function. Moreover, we could introduce a parameter that works as a threshold: if the anomaly score is below the threshold, we do not consider that situation anomalous. This structure makes sense because the policies involved in this problem are stochastic. It means that, given a state s, the policy does not return a single action to be performed, but for each action there is a certain probability of being taken: we denote with $\pi(a|s)$ the probability of performing action a when in state s. Otherwise, if adopting a deterministic policy that for each state returns just one action to be performed, we would get a structure that is too rigid. So, even if the action performed by the agent and the one in the real graph are not the same in a certain state, it could be that both actions have a certain probability different than zero of being performed, meaning that we are not in the case of an anomaly. Since the probability value of taking an action given a certain state is involved in the computation of the value function, it influences also the anomaly score, resulting in a higher anomaly score when the action is highly improbable and viceversa.

3.2.3. Application of dynamic programming methods

Once that the Markov Decision Process has been properly defined, we can proceed to the application of dynamic programming methods, in particular value iteration and policy iteration. Since this step is quite standard, in order to clarify it we will provide a very simple example, where the dimensionality of the state space is very small to allow a graphical representation.

The sequence considered will be the following:

We have 4 nodes: this means that there are 64 states and 13 possible actions. In each state it is potentially possible to perform all the actions, except for removing an edge that is not present and adding an edge that is already present. If the agent tries to perform actions that are not allowed, the situation remains unchanged. To apply dynamic programming methods, we need a perfect model of the environment as Markov Decision Process. In order to extrapolate the normal behavior, we assume that we have a clear sequence of the temporal graph that is long enough. Usually, by long enough we intend that the sequence consists of a number of steps equal to 5%-10% of the total number of states, but in very small graphs like this one it is not sufficient. In our case we suppose that the sequence has a length of 20 steps, that consists in the two times repetition of the sequence reported in the picture.

Given this clear sequence, we understand what is the normal behavior of the graph. For simplicity, we give a name to the states: we call s_1 the state at time step $t = 1$, s_2 the state at time step $t = 2$, and so on, up to state s_{10}. For what concerns the other states, we do not name precisely since we will not need to reference them. We name a_1 the action of going from s_1 to s_2, a_2 the action of going from s_2 to

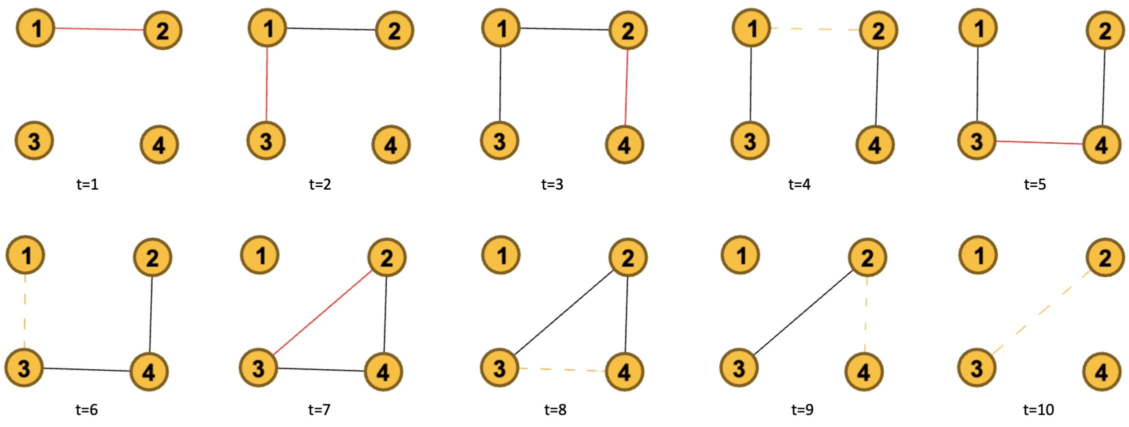


Figure 3.4: Example of a temporal graph.

s_3, up to a_9. Due to the smallness of the net, we can write explicitly the dynamics of the function, that is $p(s', r|s, a)$, the probability of going to state s' with a reward of r starting from state s and performing action a.

$$\begin{aligned}
p(s_2, 1|s_1, a_1) &= 1 \\
p(s_3, 1|s_2, a_2) &= 1 \\
p(s_4, 1|s_3, a_3) &= 1 \\
p(s_5, 1|s_4, a_4) &= 1 \\
p(s_6, 1|s_5, a_5) &= 1 \\
p(s_7, 1|s_6, a_6) &= 1 \\
p(s_8, 1|s_7, a_7) &= 1 \\
p(s_9, 1|s_8, a_8) &= 1 \\
p(s_{10}, 1|s_9, a_9) &= 1 \\
p(s_1, 1|s_{10}, a_{10}) &= 1
\end{aligned}$$

In all the other cases, p is equal to 1 when the reward is equal to 0, otherwise p is 0. Now we can try to apply the policy iteration and the value iteration methods.

In policy iteration we start with arbitrary policy and value function, and perform in loop two subsequent steps: policy evaluation and policy improvement, as long as one optimal policy is found. Value iteration can be written as a simple update operation that combines the policy improvement and truncated policy evaluation step, that is stopped after just one update of each state. The results obtained are the same from both methods: the optimal policy advises to perform a_1 in s_1, a_2 in s_2 and so on, while the highest values of the optimal value function are found in states s_1, s_2, ..., s_{10}, all as expected.

Once one optimal policy and the optimal value function have been found, it is possible to test the performance of the detector. In order to do so, we add manually some anomalies and then verify if the detector is able to find it, as we will see in Chapter 4.

3.3. Practical uses

The anomaly detection problem is a significant task and, due to its practical importance, has numerous high-impact applications in different areas. The growing attention in the past years towards spotting anomalies in data that are inter-dependent has lead to a wide range of new techniques to accomplish this task. Due to the usefulness of these methods, in this section we try to list the most common graph-based detection applications in the real-world, following the classification made in [3].

3.3.1. Anomaly detection in Telecommunication networks

We start with anomalies in telecommunication networks: there are many types of telecommunication frauds, but one of the most prevalent is known as the subscription fraud. In this case the fraudster acquires an account using false identity with the intention of using the service for free and not making any payments. In this case the nodes of the graph are the phone accounts, and the set of edges is made by the calls between them. What has been found is that fraudulent phone accounts are usually linked, since fraudsters either directly call each other or they call they call the same phone numbers, and also that it is possible to spot new fraudulent accounts by the similarities of their net, due to the fact that detected and disconnected fraudsters by the phone operator create new accounts that exhibit similar calling habits.

Type of network	Telecommunication
Nodes	Phone accounts
Edges	Calls between phone accounts
Attributes	-
Anomaly	Use of false identity without paying the service

Table 3.1: Summary table for anomaly detection in telecommunication networks.

3.3.2. Anomaly detection in Auction networks

Another common type of anomaly is the one in auction networks: auction sites, such as eBay, are attractive target for fraudsters. The majority of online auction fraud occur as non-delivery fraud, that is the seller does not deliver/ship the purchase to the buyer. In this case the nodes of the graph are the users, the directed arcs represent the transactions from the person who sells to the person that buys. The main reason for using graph-based methods in this case is that it is not sufficient to structure the detection only on the individual's features (like age, location, and so on) that are easy to fake. The fraudsters have only a local view of the net in which they operate, so it is harder for them to fit in without generating any suspect. Besides, the analysis of the fraudsters' behavior reveals that in order to manipulate the feedback and reputation system, fraudsters create additional accounts that have the task of boosting their reputation. Thanks to the behavior study, it is possible to define what is the usual pattern followed by a fraudster, allowing us to model what is considered normal and what not and to apply our method.

Type of network	Auction
Nodes	Users of auction sites
Edges	Transactions from the person who sells to the one who buys
Attributes	Users' age, geo-location, login times, session history (nodes)
Anomaly	Non-delivery fraud

Table 3.2: Summary table for anomaly detection in auction networks.

3.3.3. Anomaly detection in Financial networks

Relational learning has been used also in security fraud detection, where where the task is to find those security brokers that will probably commit violations of securities regulations in the future. Besides using information intrinsic to the brokers, such as the number and type of past violation, it is useful to introduce also relational information, such as social, professional, and organizational relationships among the brokers. In this application the likely of committing frauds depends highly on social phenomena, so this is the reason why having a graph-based approach is fundamental and gives a more complete view on the situation.

Type of network	Financial
Nodes	Security brokers
Edges	Social, professional, and organizational relationships among the brokers
Attributes	Type of relationship (edges), number and type of past violations among the brokers (nodes)
Anomaly	Violations of securities regulations

Table 3.3: Summary table for anomaly detection in financial networks.

3.3.4. Anomaly detection in Trading networks

Anomalies in financial trading network consists in a group of traders that trade among each other in certain ways so as to manipulate the stock market. In particular, those traders may perform transactions on a specific stock among themselves for a certain amount of time, during which the overall shares of the target stock in their trading accounts increase, and they produce a large volume of transactions on this stock. After the prices has increased, the traders start selling the shares to the public. The behavior of the group of traders is faced with in graph-based terms. In the first stage, where they perform transactions among themselves, the in-link weights are expected to be quite high, while in the second stage, where they sell the shares to the public, the out-link weights highly exceed the in-link ones. These fraudulent patterns are formally defined in graph-theoretic terms, so it is possible for us to easily apply our method.

Type of network	Trading
Nodes	Traders
Edges	Transactions on stocks
Attributes	-
Anomaly	Increasing falsely the values of stocks in order to sell them with a higher price

Table 3.4: Summary table for anomaly detection in trading networks.

3.3.5. Anomaly detection in Social networks

One group of malware detection methods focuses on social malware in social networks such as Facebook, the so called socware. A socware is any post appearing in one's feed in social media platforms that (1) leads the user to malicious sites, (2) makes the user perform certain actions with the promise of a false reward, (3) makes the user like or boost the reputation of certain pages without knowing, (4) makes the user replicate that the post in an unconscious way. Clearly, one way to keep track of these anomalies is by checking how the posts replicate, how much the likes of a page increase in a certain amount of time, the likes and comments under the posts and so on, in addition to other content-based features.

Type of network	Social
Nodes	Users of social networks
Edges	Interactions between the users
Attributes	Activity of the users, as number of likes, visited pages (nodes)
Anomaly	Lead the users to commit actions unconsciously, as visit malicious sites

Table 3.5: Summary table for anomaly detection in social networks.

3.3.6. Anomaly detection in Computer networks

The last application that we present is related to anomalies in computer networks, in order to detect cyber-attacks and intrusions. In this graph, the nodes represent the agents in the networks, such as file/directory servers and client nodes, while the edges represent their communication over the network, and the possible edge weights capture the volume or frequency. The assumption when tracking the dynamic nature of the network graph is that the communication behavior of a compute node changes when there is an attack. It is necessary to consider also the relational characteristic for three main reasons: (1) given the large number of compute nodes, it is impossible to monitor them individually; (2) the behavior of the nodes may depend on the behavior of the others, so monitoring them singularly could lead to lose information; (3) due to the large number of edges, it is impractical to study the highly dynamic time-series of communications volume in tandem.

Type of network	Computer
Nodes	Agents in the networks, such as file/directory servers and client nodes
Edges	Communication over the network
Attributes	The volume or frequency of the communication (edges)
Anomaly	Attacks to the network

Table 3.6: Summary table for anomaly detection in computer networks.

These are just some of the possible applications of anomaly detection in temporal graphs, but there are many others. Usually, there exist methods that are application-dependent, that rise to be applied in specific situations. Our method is more general, but there are two main obstacles: one is that we need a perfect model of the environment, the other is that the method is not suitable to deal with very large graphs. The first problem can be overcome, thanks to behavioral analysis and known fraudulent patterns that have already been found. The second is a limit of our method, related to dynamic programming: using approximate reinforcement learning methods it is possible to find a more flexible solution, that does not have issues in dealing with graph of big dimensions.

4 Implementation details and experimental protocol

This chapter contains the implementation of the method formalized in the report and the tests made to evaluate its performance; the programming language chosen is Python. The first step is the generation of the clean temporal sequences that will be used to train the agent: we explain how they are created and which are their characteristics. Given a temporal graph without anomalies, we need to extrapolate from it the dynamics of the normal behavior and to model the problem into a Markov Decision Process in order to apply dynamic programming methods. The following step consists in the training of the agent through the algorithms explained in the previous chapters. Once the detector is ready, we test its abilities in recognizing anomalies and in having learned the normal behavior. What we do is give a practical realization of the method, trying to understand its limits and possible improvements. Moreover, we try to practically go beyond the supervised setting, with the implementation of a different kind of experiment.

4.1. Dynamic Graphs Generation

As underlined several times, one of the main limits of dynamic programming in reinforcement learning is computational: if the input is too big in dimension, the computation becomes infeasible. Besides, our method requires a sequence of the temporal graph without anomalies: since it can be hard in the real world to find a temporal graph with both these characteristics, we have decided to use synthetic graphs. By synthetic graphs we intend graphs that do not come from real-world applications, but that have been generated ad hoc. Clearly, this choice comes from a necessity, but also to simplify the approach to the first implementation of the method.

Graph at time t=0

As said, the idea is that the length of the sequence is around 5%-10% of the total number of states $2^{\frac{n(n-1)}{2}}$, with n number of nodes of the graph. The graphs considered have to be quite small in dimension, but the number of nodes is chosen in a way to reach a compromise between keeping the dimension small and having a consistent sequence of the clean temporal graph. If the number of nodes is very big or very small, we need to personalize the assumption on the length of the sequence to make it practically useful. We suppose that the nodes are uniquely labelled at time $t = 0$, and that they remain the same throughout the whole sequence.

At time $t = 0$ the graph is generated by one of the two methods exposed in section 2.5, that are Erdős–Rényi and Stochastic Block Model. To recall these models, we make a short theoretical recap.

- Erdős–Rényi model: in input are given the number of nodes N and the probability p of edge existence between nodes. The vertices are linked at random with probability p to form a graph that belongs to the set of graphs $G_{N,p}$.
- Stochastic Block Model: it tends to produce graphs containing communities, subsets characterized by being connected with one another with particular edge densities. It takes the following

parameters: the number N of vertices; a partition of the vertex set $\{1, \dots, N\}$ into disjoint subsets $\{C_1, \dots, C_r\}$ called communities; a symmetric $r \times r$ matrix P of edge probabilities. The edge set is then sampled at random as follows: any two vertices $u \in C_i$ and $v \in C_j$ are connected by an edge with probability P_{ij}.

We will generate different starting graphs for each model, giving in input different parameters.

Graph sequence

Once we have generated the random graph with one of the two methods, that is the starting point of our sequence, we can proceed to produce the remaining part of the temporal graph. Logically, the sequence has to follow the same rules of the starting graph: if the Erdős–Rényi model is chosen, the edges will be added randomly following the probability p. In the same way, if the Stochastic Block Model is chosen, it is necessary to keep in mind the existing communities and probabilities of arc formation when adding edges.

The sequences are modeled in a precise way: each time step corresponds to an action, that is the addition of an arc, the removal of an arc, or no modification. When there is no modification, we intend that for 5 seconds nothing happens in the graph. Once the sequence is created, we are ready to train the agent. Clearly, we produce more than one sequence in order to make more experiments and understand better the performance of the agent.

4.2. Training and testing of the agent

After having generated the clean temporal sequences, we need to extract from them the dynamics of the environment. As recalled several times, in fact, to be able to apply dynamic programming methods in reinforcement learning we need to set the problem as a Markov Decision Process. We need to formalize the state set S, the action set A, the reward function, and the dynamics p.

4.2.1. Extraction of the dynamics

Recalling what explained in 3.2, we say that the space of the states S is composed by all the possible configurations of the graph, the space of the actions A is made by the actions add and remove an arc (both repeated for each edge) plus the action in which nothing is modified. For what concerns the reward function and the dynamics, we need to structure them relying on the clean temporal sequence. We suppose that the reward set R is made by $R = [0, 1]$. Basically, what we do is: given a state s_i that appears in the clean sequence, we define the set S_{s_i}, that is composed by all those states that in the clean temporal sequence are reached from s_i. Since the sequence is modelled in a way that between one step and the other only one action is performed, we denote with a_{ij} the action that leads from state s_i to state s_j, where s_i and s_j are two consecutive steps in the temporal graph. We define the dynamics for those states s_i that appear in the clean sequence in the following way:

$$p(s_j, 1|s_i, a_{ij}) = 1, \qquad s_j \in S_{s_i};$$

$$p(s_j, 0|s_i, a_{ij}) = 0, \qquad s_j \in S_{s_i};$$

$$p(s_k, 0|s_i, a) = 1, \quad \forall a \in A, \quad \forall s_k \notin S_{s_i}.$$

Due to the large number of states, it can happen that the agent encounters states that are not in the clean sequence: in this case the situation is a bit more complicated, since we do not have information about those states. The agent cannot mark as anomalous every situation that it does not encounter in the clean sequence, otherwise we would have a too rigid structure. In these cases, the detector behaves in the following way:

$$p(s_j, 0|s_m, a_{mj}) = 1, \quad s_j \in S_{s_m}.$$

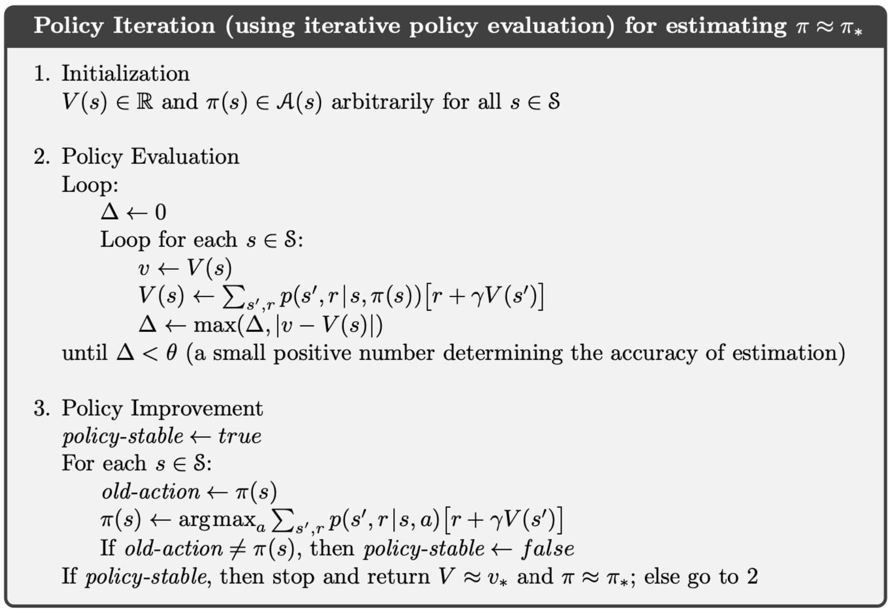
We denote with s_m a state that does not appear in the normal behavior, S_{s_m} is the set of all the states reachable from s_m, a_{mj} is the action that leads to the configuration s_j starting from the configuration s_m.

The stochastic policy that comes out from this configuration will be set in the following way:

- if the state s appears in the clean sequence, the policy $\pi(a|s)$ will be equal to the number of times that the action a is performed in state s divided by the number of times the state s appears in the sequence;
- if the state s does not appear in the clean sequence, than every allowed action from that state will be equiprobable in the policy.

Policy iteration

In order to understand how the policy iteration method is implemented, we insert a formulation of its pseudo-code taken from [2].



```
Policy Iteration (using iterative policy evaluation) for estimating $\pi \approx \pi_*$

1. Initialization
   $V(s) \in \mathbb{R}$ and $\pi(s) \in \mathcal{A}(s)$ arbitrarily for all $s \in \mathcal{S}$

2. Policy Evaluation
   Loop:
       $\Delta \leftarrow 0$
       Loop for each $s \in \mathcal{S}$:
           $v \leftarrow V(s)$
           $V(s) \leftarrow \sum_{s',r} p(s',r|s,\pi(s))\big[r + \gamma V(s')\big]$
           $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
   until $\Delta < \theta$ (a small positive number determining the accuracy of estimation)

3. Policy Improvement
   $policy\text{-}stable \leftarrow true$
   For each $s \in \mathcal{S}$:
       $old\text{-}action \leftarrow \pi(s)$
       $\pi(s) \leftarrow \arg\max_a \sum_{s',r} p(s',r|s,a)\big[r + \gamma V(s')\big]$
       If $old\text{-}action \neq \pi(s)$, then $policy\text{-}stable \leftarrow false$
   If $policy\text{-}stable$, then stop and return $V \approx v_*$ and $\pi \approx \pi_*$; else go to 2
```

Figure 4.1: Pseudo-code of policy iteration take from [2].

The pseudo-code reflects the structure of the method: we start with an arbitrary policy and an arbitrary, and then we perform the two steps of policy evaluation and policy improvement. In policy evaluation the parameter θ gives a stopping criteria: when the difference in absolute value between the old value function and the new one is smaller than θ, then we can end the loop. Unless θ is equal to zero, through this implementation of policy iteration we are able to find an approximation of one optimal policy, with a degree of closeness to the exact one that varies according to the choice of θ.

We need to pay attention to one important aspect: as anticipated, the policy in our case is stochastic. The structure of the pseudo-code remains the same, but it is necessary to take into account that there can be more than one action that maximizes the value function, and all of them need to be included in the policy with different probabilities of being taken.

Value iteration

For the value iteration algorithm, as before, we insert a formulation of its pseudo-code taken from [2].

As before, we start with an arbitrary value function V: we perform the policy evaluation and the policy improvement, with the difference that policy evaluation is truncated at the first step. For this reason, even if the results of these two algorithms are almost the same, in value iteration there is less computational

Value Iteration, for estimating $\pi \approx \pi_*$

```
Algorithm parameter: a small threshold $\theta > 0$ determining accuracy of estimation
Initialize $V(s)$, for all $s \in \mathcal{S}^+$, arbitrarily except that $V(terminal) = 0$

Loop:
|   $\Delta \leftarrow 0$
|   Loop for each $s \in \mathcal{S}$:
|       $v \leftarrow V(s)$
|       $V(s) \leftarrow \max_a \sum_{s',r} p(s',r|s,a)\big[r + \gamma V(s')\big]$
|       $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
until $\Delta < \theta$

Output a deterministic policy, $\pi \approx \pi_*$, such that
    $\pi(s) = \mathrm{argmax}_a \sum_{s',r} p(s',r|s,a)\big[r + \gamma V(s')\big]$
```

Figure 4.2: Pseudo-code of value iteration taken from [2].

effort. Also in this case we need to remind that the policy is stochastic, making the same considerations as before.

4.2.2. Detector Testing

When the agent is trained, we can test its capabilities: in order to understand the actual performance of the detector, we extract from the training sequence couples of consecutive snapshots. Given this set of couples, we modify some of the second snapshots: what we do is introducing some anomalies by adding or removing edges in positions where they are not expected or expected to be. So, we get these couples of consecutive snapshots, some of them have been modified, others not. To evaluate the goodness of the performance, we make statistics about how many anomalous situations have been detected compared to how many anomalous situations had been inserted. Moreover, we collect statistics also about those couples that have not been modified, evaluating if the agent has learned the normal behavior or not. Let us make an example considering the graph presented in 3.2.3.

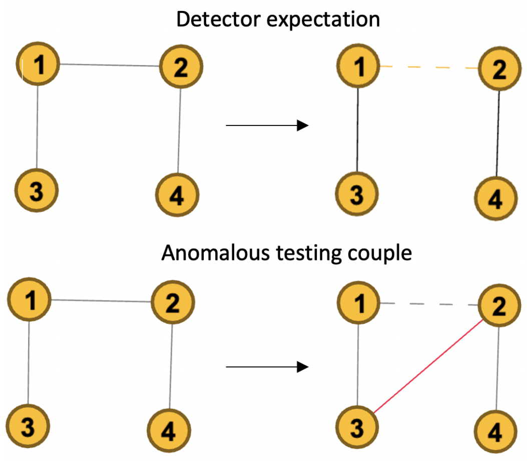


Figure 4.3: Example of anomaly.

In the picture, we have a difference between the expectation of the detector and what happens in reality. The couples made by two consecutive snapshots is made by considering the graph at time $t = 3$ and $t = 4$, modifying the second one. According to the normal behavior learned by the agent, the action to perform is the removal of the edge (1,2), while actually there is the adding of the edge (2,3). In these cases an anomaly is reported, as it should be.

Another example is reported in the following picture:

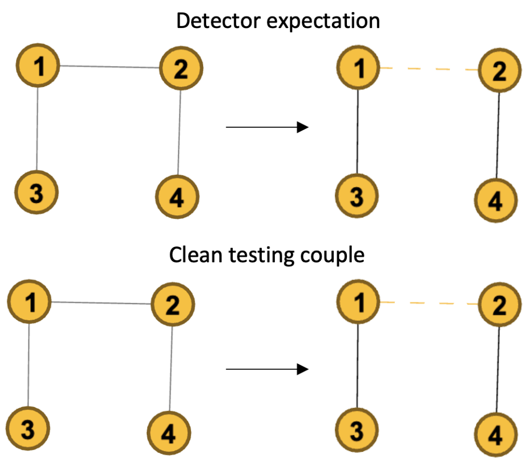


Figure 4.4: Example of normal behavior.

The couple of consecutive snapshots is the same as before, but this time it has not been added any anomaly. This kind of testing is made to understand if the agent has learned the normal behavior, and in this specific case we can give a positive answer since there is coincidence between the choice of the detector and the clean testing couple.

4.3. Experiments

In this section is presented the implementation of the algorithm developed in the report and the experiments made, specifying the parameters chosen for each instance of the experiment and the obtained results. First, we give a structure of the general procedure followed to carry the experiment, then we report the particular examples made and their results.

4.3.1. Experimental Protocol

Each repetition of the experiment is based on a different synthetic temporal graph, that is generated as explained in the previous section. We try to keep a balance between graphs generated following the principles of the Erdős–Rényi model and graphs generated following the principles of the Stochastic Block Model. In order to keep the explanation of the structure as general as possible, we will specify the parameters chosen only at the end.

One instance of the experiment is carried in the following way:

1) The first thing to do is generate the starting graph of the sequence, so we fix the number of nodes n of the graph and a random model is chosen between Erdős–Rényi model and SBM: the starting graph is ready.

2) We now need to create the remaining part of the sequence: the criteria to produce it need to respect the principle of the generative model chosen, and the length of the sequence l is equal to the 5%-10% of the number of states $2^{\frac{n(n-1)}{2}}$. Clearly, we can decide to generate a sequence where l is 10% of the number states and then truncate it according to the necessities, in order to understand how the performance changes when the sequence is shorter or longer.

3) We formalize the problem as an MDP and train the agent on the sequence using value iteration and policy iteration (the results should coincide). What we want in output is an optimal stochastic policy and the optimal value function, learned on the sequence that represents our normal behavior.

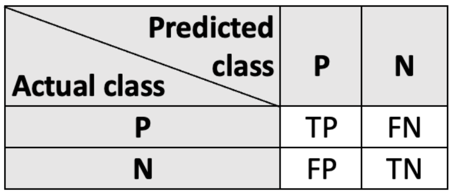
4) After the detector is ready, we evaluate its capabilities. We want to understand both if it has learned the normal behavior and if it is able to detect anomalies.

   1) We extract from the sequence couples of consecutive snapshots.
   2) We modify the second snapshot of 25% of the couples, that is we insert an anomaly, and we leave the remaining 75% as it is, because by definition an anomaly is something rare. The couples to modify are chosen randomly but maintaining this proportion.
   3) We observe how the agent behaves on the test couples, if it has understood the normal behavior and if it is able to detect anomalies, collecting the statistics of its performance.

Once this kind of procedure has been repeated several times, we aggregate all the statistics together trying to produce a meaningful analysis. The time computational complexity of one cycle is $O(m|S|^2|A|)$, where m is the number of rounds in value iteration / policy iteration.

In particular we have decided to start choosing $n = 5$, so the graphs will have 5 nodes, which means that the total number of states will be 1024 and the total number of actions 21. The length l of each sequence will be around 10% of the cardinality of the state set S, that is $l = 102$. By creating sequences of this length, we are able to cut them and consider also shorter sequences, to expand our analysis. The discount factor γ is 0.5, and the stopping criteria θ is equal to 10^{-2}. We will conduct 34 repetitions of the experiment, that is we will create 34 different temporal graphs, 17 generated following the Erdős–Rényi model, the other half following Stochastic Block Model. For what concerns the couples of consecutive snapshots, there are two ways of selecting them: we could consider the whole sequence, meaning that we consider consecutive couples of snapshots generated moving one step forward in the sequence. The other option consists in considering not all the possible couples, but half of them, by selecting consecutive couples of snapshots moving two steps forward in the sequence.

To evaluate the goodness of the detector we have decided to report the confusion matrix, or error matrix, a table that allows the visualization of the performance of an algorithm. It has the following form:



Predicted class / Actual class	P	N
P	TP	FN
N	FP	TN

Figure 4.5: Example of confusion matrix.

In our case the possible classes are two: anomalous (P = positive) and not anomalous (N = negative). By confronting the results of the predicted class and that of the actual class, four different combinations come out:

1) True Positive (TP): when there is actually an anomaly and the agent detects it.

2) False Negative (NP): when there is an anomaly but the agent is not able to detect it.

3) False Positive (FP): when there is not any anomaly but the agent detects one.

4) True Negative (TN): when there is not any anomaly and the agent thins everything is normal.

Using these elements it is possible to compute other measures to assess the performance of the method, under the condition that the set used to perform the testing is balanced between anomalous and not anomalous cases. The measures we will compute are accuracy (ACC), recall (true positive rate - TPR) and specificity (true negative rate - TNR).

$$ACC = \frac{TP + TN}{P + N},$$

with $P = TP + FN$ and $N = TN + FP$. Accuracy is the number of correctly predicted data points out of all the data points.

$$TPR = \frac{TP}{P} = \frac{TP}{TP + FN} = 1 - FPR,$$

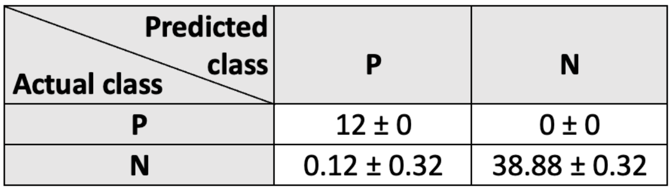
with FPR false positive rate. The true positive rate refers to the proportion of those data points that received a positive result on the test out of those that actually have the condition.

$$TNR = \frac{TN}{N} = \frac{TN}{TN + FP} = 1 - FNR,$$

with FNR false negative rate. The true negative rate refers to the proportion of data points that received a negative result on the test out of those that do not actually have the condition.

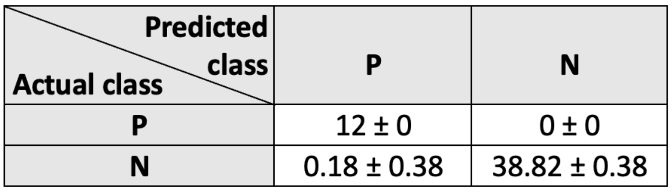
4.3.2. Results of the experiment

We now report the results of the experiment, considering separately the 17 sequences generated according to Erdős–Rény model, and the 17 according to Stochastic Block Model. The testing set used for each sequence is composed by 39 normal couples and 12 anomalous couples. Below we can find the confusion matrices for the two cases, reporting for each entry the mean value and its standard deviation.



Predicted class / Actual class	P	N
P	12 ± 0	0 ± 0
N	0.12 ± 0.32	38.88 ± 0.32

Figure 4.6: Confusion matrix for the ER sequences.



Predicted class / Actual class	P	N
P	12 ± 0	0 ± 0
N	0.18 ± 0.38	38.82 ± 0.38

Figure 4.7: Confusion matrix for the SBM sequences.

As expected, the mean is near to the optimal results. The standard deviation is a measure of the amount of variation or dispersion of a set of values: a low standard deviation, as in this case, indicates that the values tend to be close to the mean of the set. This measure confirms the consistency of the results of the experiments.

Now we can compute accuracy, true positive rate and true negative rate, using the values in the mean table. We start with the ER sequences:

$$ACC_{ER} = 0.99$$

$$TPR_{ER} = 1 \quad FPR_{ER} = 0$$

$$TNR_{ER} = 0.99 \quad FNR_{ER} = 0.01$$

We proceed with the SBM sequences:

$$ACC_{SBM} = 0.99$$

$$TPR_{SBM} = 1 \quad FPR_{SBM} = 0$$

$$TNR_{SBM} = 0.99 \quad FNR_{SBM} = 0.01$$

It is clear that the performance of the method is optimal, both in understanding the normal behavior and in detecting anomalies, whatever is the generative model chosen.

For what concerns the performance of the policy iteration and value iteration algorithms, it is interesting to notice that in all the experiments we have 6 iterations for policy iteration, in which the policy evaluation step takes around 8 iterations each time, and 8 iterations for value iteration. However, stating that both give the same results, the value iteration algorithm is faster, thanks to the truncated policy evaluation.

4.3.3. Towards an unsupervised setting

Another type of experiment that we can try to perform moves the report towards a more unsupervised setting. The assumption of having a clean temporal sequence can be limiting sometimes: for this reason, now we simply suppose to have a temporal sequence, not free from anomalies. We want to exploit the fact that, by definition, an anomaly is something rare, that does not happen often: we can define as anomalous those couples of states and actions that appear a number of times less or equal than a certain fixed threshold. Since we are detecting anomalies on edges, what we mark as anomalous is the edge that is added/removed that leads to the anomalous state. For this reason, what results anomalous is performing a certain action on a certain state that leads to the anomalous snapshot: now we have to consider the possible combinations between all the states and all the actions.

Recalling that the policies involved in problem are stochastic, we can define the threshold mentioned before in terms of probability: for each state, those actions that have a probability of occurring that is smaller or equal than the threshold are marked as anomalous actions. Since the policy is involved in the computation of the value function, this concept is naturally present in the non-binary ranking score defined in 3.2.2. This experiment may result more interesting than the previous one, since now the anomalies can be found directly in the probabilities computed by the stochastic policy.

In order to carry this type of experiment, we need longer sequences than before: according to the length of the sequence and to the total possible number of states we will fix a different threshold. The procedure to generate the sequence is the same as before: we choose one generative model, fixing all the parameters, create the sequence according to the model chosen, and then train the agent. This time it is not necessary to create testing couples, since the anomalies will be extrapolated directly from the policy learned by the agent.

Differently than before, we use graphs with 4 nodes, and sequences of length 2000, in order to reach a compromise between length of the sequence and speed of computation. Since the number of nodes is 4 now, we restrict our experiments to the Erdős–Rényi model, since the Stochastic Block Model requires at least five nodes to function properly. We need to remember that in this application the total possible combinations of states and actions is equal to $|S| \cdot |A| = 64 \cdot 13 = 832$. We will use only value iteration to train the agent in order to speed up the experiments. The threshold that we set is state-dependent: for each state s we say that performing action a in s is anomalous if the probability $\pi(a|s)$ according to the optimal policy π is less or equal than $\frac{1}{\#s}$, where $\#s$ is the number of time that state s appears in the sequence. It means that the couple state-action is considered anomalous if it appears one time in the sequence. Also in this experiment we need to be careful in choosing the threshold according to the length of the sequence examined. In this case we do not have to clean the optimal policy, we simply have to add the condition that the agent marks as anomalous those actions that have a probability of occurring smaller or equal than the threshold chosen. We now report the results of the performed tests:

As expected, the results are optimal: the agent detects all the anomalies.

#Sequence	#Anomalies - real	#Anomalies - detected
1	27	27
2	22	22
3	23	23
4	17	17
5	16	16
6	28	28
7	33	33
8	16	16
9	21	21
10	22	22

Table 4.1: Performance of the second experiment.

5 Conclusions and Future Directions

The method presented in the report is just a small step in what could become a bigger work. The aim of this chapter is to give a general overview of the algorithm formalized and its results, proposing the possible developments and improvements. This project has the aim to realize a detector on temporal graphs that is flexible and capable of generalization, able to detect anomalies not only in the case of presence of an unexpected elements but also in terms of absence of an attended element. Considering the growing attention towards dynamic nets and anomaly detection, and the multitude of useful real-world applications, we think that this work fits well in the current scientific research panorama.

5.1. Final considerations

We now give a complete overview of the method developed in the report, taking into account the results of the practical experiments and the problems encountered. We are able to look at the development of the algorithm in its completeness: we have started from the conceptual background of anomaly detection in temporal graphs and of reinforcement learning, proceeding to the theoretical formalization of the problem, up to the practical realization of the method. All these elements allow a full comprehension of the work, that we will recap and comment in the following paragraphs.

5.1.1. Report summary

As the title itself of the report says, the main elements involved in this work are three: anomaly detection, temporal graphs and reinforcement learning techniques. We have briefly presented these three macro-arguments in the first chapter, trying to report the fundamental concepts and latest developments. The second chapter represents the first step towards the realization of the method, because it is dedicated to its formalization. We have defined what are the characteristics of our input temporal graph: the sequence must be without anomalies, modelled in a way that between each couple of consecutive time steps only one action is performed. Then, another fundamental step is the Markov Decision Process setting: we have explained how to transform the input sequence into the tuple (S, A, p, R). The last element that requires a formal definition is the anomaly itself: we have stated what we intend by anomaly, that is the situation in which the trained agents expects certain situations to happen but actually they do not. The anomalies require also a function score, that is a way to measure their gravity: we have proposed two different solutions, the binary one and the non-binary one, in which the value function is involved. The use of the value function allows to take into account the fact that the policy is stochastic, that in some states there may be more than one normal action to perform, with a probability that depends on how many times the couple state-action is encountered in the clean temporal sequence.

Once that all these elements have been formalized, we have applied dynamic programming techniques, first just theoretically, and then in the third chapter through a practical implementation. We have carried two different types of experiments: the first one supposes the knowledge of a clean temporal sequence of a certain adequate length, from which extrapolate the normal behavior of the graph. As expected, being in a supervised and small world, the performance of the method is very precise and accurate, both in forecasting the normal behavior and in detecting anomalies.

The second experiment tries to go beyond the supposition that there is a clean temporal sequence, because in practice it is not always satisfiable. We exploit the fact that by definition an anomaly is something rare, that doesn't occur often. What we need is a long sequence of the temporal graph: fixing a certain appropriate threshold, we mark as anomalous those couples of states and actions that appear in a number less or equal than the threshold. Clearly, being the anomaly on arcs, what we mark as anomalous is the addition/removal of the arc that leads to the anomalous state.

In conclusion, we can list the main contributions of the report:

- the formalization of the model for the first reinforcement learning approach to detect anomalies in temporal graphs;
- the definition of a procedure to generate sequences of temporal graphs starting from a graph generative model;
- the implementation of the algorithm;
- the experimental protocol to test the performance of the method;
- the extension of the proposed approach towards an unsupervised setting.

We will talk about all the possible developments in the following section.

5.1.2. Results and limits

The experiments made in chapter 4 have allowed us to understand the performance of the method: we now provide an analysis of the results obtained.

In the first type of experiment we suppose to have a clean temporal sequence, from which the agent can learn the normal behavior. The testing couples are extracted from this sequence: some of them are modified, in order to represent the anomalies, others are untouched. As explained, in the cases in which the setting is supervised and small, the performance is expected to be optimal, as it is in this case. We have very high levels of accuracy, true positive rate and true negative rate, both computed on each single instance of the experiment and in the aggregated case. The single results of the experiments are very similar one another in terms of the confusion matrix: this is reflected in the very low value of standard deviation. Also in terms of number of iterations, in all the cases to reach the optimal policy and value function were necessary 6 iterations of policy iterations and 8 of value iteration, a quite reasonable number of repetitions.

Since the performances are so high, we can say to have built a detector that is very accurate in predicting the normal behavior and in finding anomalies, but we need to understand which are the underlying assumptions that led us to this results. First of all we have supposed to have a clean temporal sequence of the graph: this is often not possible in practice. Usually, in the real world we can get patterns of what is considered normal and what is not, and through these patterns it is possible to extrapolate the normal behavior. Anyway, also this condition is not always satisfied. Another aspect that can be viewed as a limit is the computational one: using dynamic programming techniques implies restricting the application on quite small graphs. We have already seen that the number of states grows exponentially with respect to the number of nodes, meaning that for big graphs it is more suitable to use approximate solution methods. This exact model is very heavy for what concerns memory and time.

The second experiment is based on a different assumption: we suppose to have a very long sequence (surely longer than the total number of combinations between states and actions), that does not have to be free of anomalies. Also in this case the detector learns the behavior from the sequence, but recalling that an anomaly is by definition something rare, we fix a threshold and mark as anomalous all those couples of states and actions that appear a number of times smaller or equal than the threshold. In this case, we can apply the threshold directly on the probabilities of occurrence of the actions, that come out from the stochastic policy.

The results of the experiment appear to be optimal, since the detector manages to find all the anomalies present. This kind of results was expected: by definition, we have set that in the optimal policy the probability of performing each action corresponds to the number of times the action is performed in the state over the total number of times the state appears in the sequence.

This method can be useful when having a temporal sequence that is not clean: the important thing is to choose the threshold carefully, because it depends strongly on the length of the sequence. The principal limit is that, since the sequence is longer than before, the algorithm results heavier computationally, both in terms of time and space.

The main drawback of the setting of the second experiment is that we rely strongly on the definition of anomalies, that is on the fact that an anomaly is something rare: in practice, it is not always this simple.

5.2. Possible developments

In this work we have considered the most basic case possible, that is the one of small undirected temporal graphs, of which we know a sequence that describes the behavior without anomalies. Moreover, since it is a first approach of the anomaly detection problem in temporal graphs with reinforcement learning techniques, we have decided to apply dynamic programming methods. As mentioned several times in the report, this is just a first step towards the realization of better and more flexible solutions: the possible developments are many.

5.2.1. Small temporal graphs

One possible path to follow is keep working on small temporal graphs, that is when the state and action spaces are small enough to be represented as arrays or tables. Besides dynamic programming algorithms, in section 2.3.2.1 we have introduced also Monte Carlo methods and temporal-difference learning. Both these methods could result more useful in some real-world applications, since they both do not require a model of the environment.

Monte Carlo methods learn value function and optimal policies from experience in the form of sample episodes. The tasks need to be episodic to assure that well-defined returns are available, because the methods are ways of solving the reinforcement learning problem based on averaging sample returns. Due to the sample setting, it is not possible to apply them in all the cases, and when it is there is the necessity to formulate the problem in a different way. What is required is only experience, in the form of sequences of state, actions and rewards from actual or simulated interaction with the environment. In our case it means that we still need the agent to interact with an environment that is clear, without anomalies. The ideas are the same already exposed for dynamic programming, but extended to the Monte Carlo case in which only sample experience is available.

We distinguish two types of approaches: the on-policy one and the off-policy one. This differentiation comes from the dilemma that all learning control methods encounter: they seek to learn the optimal behavior, but they need to behave non-optimally in order to explore all the actions. The on-policy approach, that is the one we have already used in DP, learns the action-values for a near-optimal policy that still explores. In the off-policy learning instead two policies are used: one that is learned and that becomes optimal (the target policy), and another that is used to generate the behavior (behavior policy). Off-policy learning is more complex and slower to converge, but it is more powerful and general.

Monte Carlo methods represent a step forward with respect to dynamic programming because we can learn directly from experience (simulated or real): in DP all the probabilities must be computed before the algorithms can be applied, and those computations are often complex and error-prone. Differently, generating the samples required by Monte Carlo methods is much easier. One limitation is surely the fact that the tasks need to be episodic, which is not always fully appropriate for our particular application temporal graphs.

Combining Monte Carlo ideas and dynamic programming ideas we get to temporal-difference learning, a central and novel aspect of reinforcement learning. In fact, these methods sample from the environment, like Monte Carlo methods, and perform updates based on current estimates, like dynamic programming methods. While Monte Carlo methods only adjust their estimates once the final outcome is known, TD methods adjust predictions to match later, more accurate, predictions about the future before the final outcome is known: this is a form of bootstrapping. Temporal-difference learning has an advantage over dynamic programming because it is not required the knowledge of the dynamics of the environment; the advantage over Monte Carlo methods is that they are naturally implemented in an online incremental way, still assuring convergence.

The methods presented represent possible developments of our algorithm, but still we need a certain knowledge about the normal behavior of the environment and the condition that the graph is small in dimension. This second constraint will be overcome in the following subsection, in order to have a wider range of real-world applications for the anomaly detection problem in temporal graphs.

5.2.2. Big temporal graphs

One important development in this project is being able to deal with big temporal graphs: in this way it is possible to treat most of the real-world cases, where the state space is combinatorial and huge. We have already seen how big the state space can become in relation to the the cardinality of the vertex set: $|S| = 2^{\frac{n(n-1)}{2}}$, with n number of nodes of the undirected graph. The situation is even more complex if the graph is direct, in fact we have $|S| = 2^{n(n-1)}$, with n number of nodes. In order to treat this case we need to use approximate solution methods, already mentioned in 2.3.2.2, that can be considered an extension of tabular methods. The aim is to find a good approximate solution using limited computational time and data: in fact, it is impossible to encounter all the numerous states, and to make relevant decisions, it is necessary to generalize from previous encounters with different states that are similar to current one. The key is to combine reinforcement learning methods with generalization methods: it means trying to build an approximation of a given function (for example the value function), taking examples from it and using them to generalize the behavior. This type of setting is particularly useful when it is possible to find patterns in the normal behavior or in the structure of the anomalies, so that when similar situations are presented it is more natural to think that the outcomes are the same.

There are several methods that we have introduced in 2.3.2.2: the one of on-policy training, in which the policy is given and only its value function is approximated; the control case, in which an approximation to the optimal policy is found; the off-policy learning with function approximation and the policy gradient methods, which approximate the optimal policy directly without an approximate value function. The strategy would be to try to formalize and implement all these methods and evaluating their performances, in order to understand which one performs better and in which cases.

In addition, in order to get a more complete detector, some features could be added. For example, it would be interesting to consider a temporal feature: it could happen, in fact, that a certain event needs to be considered anomalous in some situations and normal in others. For example, we can think about an e-commerce site: when a new product is launched, it is expected that the number of purchases or visits to the site increase in that period; in normal situations, however, it could cause suspects. Consequently, the agent should be aware of the temporal moment in which it is when detecting anomalies. Once that an anomaly is detected, it is important to understand also where it comes from and the reasons that have caused it. We can think of a DOS cyber-attack, in which a device is targeted and put out of use because a lot of devices try to connect with it (in a graph it means that there are a lot of arcs coming from different nodes that are all directed to a single node). If the agent would be able to detect not only the anomaly but also its meaning, it would be easier and faster to remedy it. The last improvement we could make is about the formalization of the setting of the problem: instead considering each time step as an action, we could sample the graph sequence in actual temporal steps of the duration of a certain number of seconds, in order to have a more complete view of the graph. In this case we would not be detecting specific anomalous edges but anomalous snapshots, being able to eliminate also the hypothesis that the set of nodes is fixed.

One thing to notice is that we still are assuming that we have some knowledge about what is the normal behavior: in real-world applications it is not always true. The biggest step forward to make is to eliminate this supposition. Clearly this development requires a well defined formalization and a massive study, but just to give some hints at which could be the possibilities we could say that one way to guide the agent in the training is to give rewards based on the level of change between one graph and the other. This means that when passing from a graph to another if the structure of the two graphs is too different the reward will be low, meaning that it is improbable to switch from one snapshot to the other. This time the difficulty lies in understanding what we mean by ’different snapshots’, that is defining a function that computes the degree of diversity between two graphs: this function is application dependent and needs to be thought carefully.

Bibliography

[1] S. Ranshous, S. Shen, D. Koutra, S. Harenberg, C. Faloutsos, and N. F. Samatova, “Anomaly detection in dynamic networks: a survey,” *Wiley Interdisciplinary Reviews: Computational Statistics*, vol. 7, no. 3, pp. 223–247, 2015.

[2] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*. MIT press, 2018.

[3] L. Akoglu, H. Tong, and D. Koutra, “Graph based anomaly detection and description: a survey,” *Data mining and knowledge discovery*, vol. 29, no. 3, pp. 626–688, 2015.

[4] S. Brin and L. Page, “The anatomy of a large-scale hypertextual web search engine,” *Computer networks and ISDN systems*, vol. 30, no. 1-7, pp. 107–117, 1998.

[5] J. Sun, H. Qu, D. Chakrabarti, and C. Faloutsos, “Neighborhood formation and anomaly detection in bipartite graphs,” in *Fifth IEEE International Conference on Data Mining (ICDM’05)*. IEEE, 2005, pp. 8–pp.

[6] C. C. Noble and D. J. Cook, “Graph-based anomaly detection,” in *Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining*, 2003, pp. 631–636.

[7] Z. Chen, W. Hendrix, and N. F. Samatova, “Community-based anomaly detection in evolutionary networks,” *Journal of Intelligent Information Systems*, vol. 39, no. 1, pp. 59–85, 2012.

[8] X. Guo and L. Zhao, “A systematic survey on deep generative models for graph generation,” *arXiv preprint arXiv:2007.06686*, 2020.

[9] C. C. Bilgin and B. Yener, “Dynamic network evolution: Models, clustering, anomaly detection,” *IEEE Networks*, vol. 1, 2006.

[10] C. Szepesvári, “Algorithms for reinforcement learning,” *Synthesis lectures on artificial intelligence and machine learning*, vol. 4, no. 1, pp. 1–103, 2010.