UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Facultat d'Informàtica de Barcelona

Barcelona Neural Networking Center

iGNNspector: A Tool for Graph Neural Network Acceleration

Bachelor's Thesis
Specialization in Computing

Nil Vidal Ràfols

Director: Sergi Abadal

June 23rd, 2021

Abstract

This thesis is also part of a bigger project that is composed of 2 other final degree thesis. The motivation behind the project was to research ways of solving the challenges that the GNN field currently faces. With GNNs, there is not a single model that can work with all types of graphs, but there exists different strategies that are tuned for every case. Also, the time and memory complexity of a GNN is greatly dependent on the type, size and topology of the graph it is applied to. This can slow down development of GNNs or make it difficult for newcomers to the field to design and deploy their algorithms.

The first thesis of this project is aimed to research the characterisation of diverse graph datasets, relevant to GNNs, in order to better understand its properties. The objective of the second thesis is to research a series of algorithms that allows for the acceleration of GNNs and a more efficient memory usage.

Finally, this thesis is aimed to propose a way to solve the current challenges in the form of a software tool that can benefit from the knowledge obtained in the other thesis. Named iGNNspector, this tool is intended to explore a way of solving these issues. It is aimed to be used by a wide range of users regarding their experience in GNNs and consists of a framework that also includes a user interface.

Contents

1. Context and scope

1.1. Context

1.1.1. Introduction

In order to give the reader the context required to understand the purpose and scope of the project, I will introduce in this section a brief description about the terms and concepts surrounding the project, the problem that the project is intended to solve, and finally, the different parties and stakeholders who are involved in or may be affected by it.

This bachelor's thesis is part of an extended project composed of three theses in total, the works of which complement each other. The project aims to study the field of Graph Neural Networks, which is a branch of Machine Learning and will be explained further on, in order to better understand the relationship between the characteristics of a given graph and the performance of different algorithms when executed with said graph. With the knowledge gathered, we also aim to develop a tool that can help researchers in the field. This last part is the one my thesis will be based on.

This work has been proposed by and is being developed with the help of the researcher Sergi Abadal and the research group he works at, BNN-UPC or Barcelona Neural Networking Center. This group was created by professors Albert Cabellos and Pere Barlet-Ros, from the Facultat d'Informàtica de Barcelona FIB, with the goals of carrying out fundamental research in the field of Graph Neural Networks and educating and training students who are interested in this field.

1.1.2. Terms and concepts

Graph Neural Networks (GNNs) belong to a branch of Machine Learning and are composed of algorithms and techniques that work with graph structured data [1].

Machine Learning is a field in computer science that studies algorithms capable of making predictions about certain data after they have been trained on samples of similar data. In the last decade, this field has experienced a massive growth and has changed entire industries due to the variety of ways it can be applied. Nonetheless, there exists data that is inherently relational, that is, different parts of a whole system are related to each other. This data can be represented as graphs, which are structures composed by nodes and edges, the strings that connect one node with another.

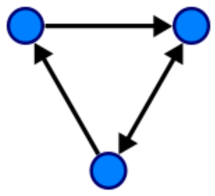


Figure 1: A directed graph [2]

A graph G can be described mathematically as:

$$G = (V, E) \tag{1}$$

where V is the set of nodes from the graph G and E is the set of edges $(v, u),\ v, u \in V$.

Some examples of this type of data could be the users of a social network (nodes) and the relationship between them (edges). Or a set of molecules that are made up by atoms bonded together in different ways. Each node from the examples above would have a *feature vector* that would store information like the type of atom or the personal information of the user.

There exists Deep Learning algorithms for almost every type of data. Convolutional Neural Networks (CNNs) are good for grid data, e. g. an image made up of pixels. Recurrent Neural Networks can deal with temporal data, etc. However, those conventional techniques struggle to achieve good results when dealing with relational data, where data points are not ordered in a certain way. Unlike pixels or discrete moments in time, relations between nodes in a graph do not follow a strict pattern. Here is where GNNs come in [3].

1.1.2.1. Structure of GNN algorithms

GNNs are a relatively new field that has experienced quite a lot of development in the last few years. Their power resides in the fact that they can infer the features of an unknown node from the characteristics of its neighbour nodes (the ones it is connected with). This process is called *Message Passing*. As the name implies, it consists in passing a message from a neighbour node through an edge to another node.

Message Passing is not only a single step process, the features from the neighbouring nodes undergo a series of operations listed below:

- A linear transformation can be applied to the *feature vector* of a neighbour node,
- Then, the transformed features of the vectors are *aggregated* together using a specific criteria.
- Finally, the *feature vector* of the target node is *updated* also using a specific criteria and maybe a linear transformation.

A *Message Passing* process can be thought of as a *layer*, and several *layers* can be executed sequentially to create a GNN.

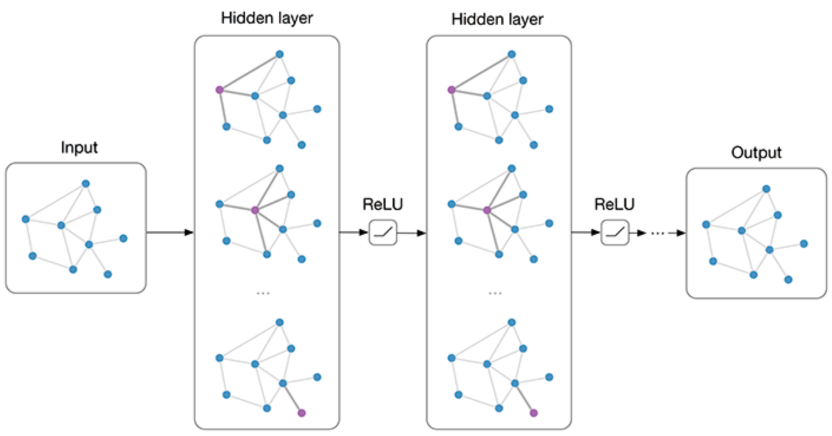


Figure 2: Layers from a GNN [4].

Next, I will explain how a couple of influential GNN models work.

Graph Convolutional Network or GCN.

One of the first papers to kick off the field of GNNs was "Semi-supervised Classification with graph Convolutional Networks", where Thomas N. Kipf et al. introduced in 2017 a model of GNN that implemented the *Message Passing* process as a convolution of the neighbouring nodes of a target node. This operation can be expressed as the multiplication of a feature vector matrix containing as many rows as nodes and the adjacency matrix of the graph. Moreover, the model adds to the multiplication a learnable weight matrix and a diagonal degree matrix to give importance to nodes with lower degree [5].

$$\mathbf{X}' = \hat{\mathbf{D}}^{-1/2}\hat{\mathbf{A}}\hat{\mathbf{D}}^{-1/2}\mathbf{X}\mathbf{\Theta}, \tag{2}$$

The Message passing equation where X and X' are the feature vector matrices, θis the learnable weight matrix and D is the diagonal degree matrix [6].

Graph Attention Networks or GAT.

In the paper "Graph Attention Networks", The authors wanted to improve in 2018 the way GNNs gave importance to the surrounding nodes. GATs are quite similar to GCNs but for one key difference. They determine the importance that a target node should give to its neighbours not by a function of their degree, but by a value that should be found through a learning process. This approach turned out to be very powerful [7].

1.1.3. Problem to be solved

Deep Learning algorithms, like CNNs or Multilayer Perceptrons are computationally expensive to train, because they have to do thousands upon thousands of calculations. The more layers or neurons a model has, the more time it will take to complete training. Nonetheless, the computational complexity of a model is constant. Once the architecture of a model is known its execution time can also be predicted.

GNNs are a special case among Deep Learning algorithms. The structure of a GNN does not depend only on the number of layers it has or the steps done in a Message Passing. Layers in a GNN have the topology of the input graph, which means that depending on the structure of the graph, the computational and memory complexity of the model can vary enormously [1]. Even further, the accuracy of a given GNN algorithm may also depend on the connectivity of the graph and the feature vectors of its nodes. Moreover, the datasets GNN researchers have to work with contain graphs with huge amounts of nodes and edges. This fact makes researchers have to spend large amounts of precious time trying to come up with the best set of configurations to run their algorithms so that they are accurate and fast.

This thesis aims to develop a software tool that can help GNN researchers circumvent the above problems in order to save a lot of time and optimize results. The hypothesis, hinted at in recent works, is that there is a correlation between the graph characteristics and the performance of a certain set of GNN algorithms , [8], To exploit this, the tool has to analyze a given graph dataset and extract a set of characteristics. With this analysis and the knowledge we will gather, the tool will infer a GNN model and a memory representation that best suits the dataset and provide it to the user, or run it directly.

1.1.4. Stakeholders

The stakeholders can be classified by their involvement in the thesis and the benefits that may bring to them. Firstly, we have the stakeholders who are directly and actively involved.

The stakeholders that will be actively involved in the project are, first of all, myself, the thesis researcher. I will be responsible for the planning, development and documentation of the software tool and its core implementation. The director, Sergi Abadal, will monitor, give advice and help steer the planning and development of the work done. Furthermore, since the thesis is a collaborative project that involves the work of multiple students, it means that myself, Cristina Tubert and Carlos Gascón will be benefiting from our respective work in the field.

Although the BNN team will not be as actively involved as the stakeholders mentioned above, they will also provide us with knowledge and tools to make the development of our work possible, like access to GPU servers able to accelerate the processing of our GNN algorithms.

Finally, there are the stakeholders that will be benefiting from the results of the work done. As stated in the previous section, this thesis has the potential to benefit GNN researchers working at BNN, and also researchers from the scientific community at large, in the form of money, time and resources saved. Users with a varying range of knowledge or experience in the GNN field can benefit from the capabilities of the tool, as well as companies interested in more efficient algorithms and faster development and deployment of GNN solutions, like Amazon, Pinterest, Alibaba, among others.

1.2. Justification

In the field of Data Science, one of the most used programming languages is Python, which has a wide range of modules that allow researchers to manipulate and analyze data. At the same time that research at GNN was growing, *Pytorch Geometric* [9] did as well. Nowadays, this module is one of the most, if not the most used module for GNN development. As a consequence, there is plenty of documentation about the module and how to use it. As it was explained before, graphs can have vastly different topology. That is why Pytorch Geometric implements different techniques to calculate each step of a GNN model.

For those reasons, the algorithms related to GNNs in this thesis will be constructed with the help of Pytorch Geometric. Nonetheless, the BNN group has not incorporated Pytorch Geometric to their research yet, as they have been using other frameworks like TensorFlow, which is another highly popular framework specialized in Deep Learning. This way, the thesis could bring more variety to the techniques used at BNN and introduce the team to Pytorch Geometric.

Although Pytorch Geometric is a powerful framework, it does not provide advice on which GNN techniques to use for the task at hand.. To accomplish that, a prior analysis of the graphs in a dataset has to be done. For that, we intend to use NetworkX [10]. As it is a Python module as well, it can be used jointly with the other modules and it brings with it a lot of methods to create and analyze graphs.

Finally, the tool will combine in its development these libraries, among others, in order to achieve the objectives of the thesis.

1.3. Scope

1.3.1. Objectives

To develop the tool that has been described before, is not precisely a short journey. It requires acquiring extensive knowledge in the field of GNNs, learning to use different frameworks and understanding how they internally work, performing testing to ensure that the hypothesis about the performance of GNNs are correct, and finally, combining all parts of the software that make up the tool and test them rigorously to make sure they work seamlessly with each other.

All of the steps of the project can be broken up into a set of objectives that must be accomplished. Those objectives can be either formation objectives or practical objectives:

- Study Graph Neural Networks in several fronts.
  - From a theoretical point of view. The general idea behind GNNs, the different operations that can be performed in a Message Passing process, and in what situations they work better. Which are the most important types of GNNs and how they are implemented.
  - From a computation standpoint. How the data about graphs is stored in memory. What representations are best suited for each type of graph. What is the cost both temporal and spatial using a concrete representation, operation, structure of the model and structure of the graph.
- Learn about Pytorch Geometric, NetworkX, Qt, and other modules that will be required.
  - Research how Pytorch Geometric works on the inside. How it performs operations, how it stores graphs, etc, in order to determine what is optimal for which type of graph.
  - Learn to program with Pytorch Geometric to test its algorithms and integrate it to the tool.
  - Learn to program with NetworkX to analyze graphs and integrate it to the tool.
  - Learn how to implement Graphical User Interfaces or GUIs with Python to be used for the tool.
- Develop and implement heuristics that can determine the set of algorithms and representations that best suits a dataset. These heuristics must try to reflect the conclusions obtained from the parts of the project developed by Cristina Tubert and Carlos Gascón, as well as

further research provided by BNN and the GNN research community and tests performed with the servers that BNN has provided and the Pytorch Geometric implementations build for the project.

- Develop a GUI for the tool.
- Develop and implement the software core where within its structure will integrate Pytorch Geometric, NetworkX, the GUI, and a simple database to store information from graphs that have already been analyzed, configurations, and such.
- Test each part of the software to make sure that they work as intended by themselves.
- Join all parts of the tool and conduct extensive testing to make sure everything works together as intended.

1.3.2. Requirements

- The tool will support a set of file types (yet to be determined) that store graph data, as well as pickle files which store instances of the graph representation class from Pytorch Geometric and NetworkX.
- The tool will store the characteristics of a graph once this is provided as input in order to avoid having to re-analyze it if it is used again.
- The tool and all of its dependencies have to be cross-platform. It has to work the same way regardless of the OS that the user might be working on.
- The external modules used within the tool, like Pytorch Geometric, must not be altered or customized to work with the tool. Doing so could make it impossible or very tedious to integrate new updates of these modules to the tool.
- The user should be able to acquire the resulting algorithms provided by the tool as pickle files or Python scripts.
- If the dataset is very big and the tool expects to complete the analysis in no less than 10 minutes, it should ask the user if she/he wants to analyse a subset of the dataset to receive quicker results. Otherwise, the benefit of using iGNNspector would be completely offset by the time required by the characterization.
- Since the tool will be written in Python, to ensure a well structured and clean code, it will mostly comply with Pep8 rules.

1.3.3. Risks and obstacles

Some setbacks could arise during the project development. We next describe some of these plausible risks:

- Deadlines auto imposed by the planning might not be met due to things not foreseen at the beginning.
- The developed heuristics might be harder to perfect than expected. Graphs are complex, and the performance of a GNN might be tight to a lot of variables. This fact could lengthen the time it takes to develop proper heuristics.
- In very large datasets, e.g. datasets that do not fit in main memory, it is possible to encounter problems with NetworkX. Other datasets which do fit in memory but are still very large might also take a lot of time to analyze with NetworkX. For these reasons, it is expected that in some cases, only a subset of the graph should be analyzed.
- It is said that the majority of the time spent developing software is fixing bugs. In order to minimize the amount of bugs and the time spent fixing them, it will be necessary to write well structured and clean code and perform a lot of testing.
- Since the thesis is part of a bigger project and is in part dependent from the findings of other student thesis conducted at the same time, an obstacle from one thesis could affect the others and slow down in some measure the development of the tool.

1.4. Methodology

The objectives required to get to the goal of the thesis, as every other project, are plenty and varied. Moreover, before a task can be considered finished, it has to transition through some phases like research, development and testing. Since the majority of the time I will be working on the project by myself, I need a method that allows me to keep track of all tasks in the easiest way possible, and bring some flexibility to the process in case an obstacle shows up.

With the points presented above, it is clear that an agile methodology is the way to go. In an agile development process requirements and solutions can be revisited as experience shows better ways to go or some walls are encountered along the way. Nonetheless, if the project is not well organized, some obstacles or revisions can hurt the planning and make it difficult to meet deadlines. That is why I intend to organize tasks in a way that they are independent from each other.

The version control tool Git, combined with the service Github will certainly help. The code will be hosted by a private Github remote repository. A branch for each part of the project, e.g. a branch for graph analysis, a branch for the GUI, etc, will be created to isolate the development in each front. This way, some parts of the code can be revisited without affecting other areas. Also, tests can be conducted without interference.

Finally, I like the concept behind the Kanban methodology, where tasks and their state are presented in a very visual way. It is easy for our brains to identify objects with a distinct color and form, and Kanban uses this fact at its advantage.

Tasks will be represented as “post-its” in a canvas, and the same color will be given to tasks from the same objective. Then, they will be placed into 5 different columns that represent their progress status. Which are:

- To do. Tasks that haven't been started yet.
- Researching. Tasks the contents from which are being researched and learned.
- Developing. The knowledge gained from the last step is used now to develop a part of the tool. Some simple testing is expected in this step.
- Testing. The developed software undergoes more complete and rigorous testing.
- Done. After every step is completed successfully, it is considered that a task is finished.

Once the tasks related to individual parts of the software are done, they will be tested jointly. For example, if some problems arise, some tasks might go back to the developing and testing columns for a moment to fix them before trying again the whole system.

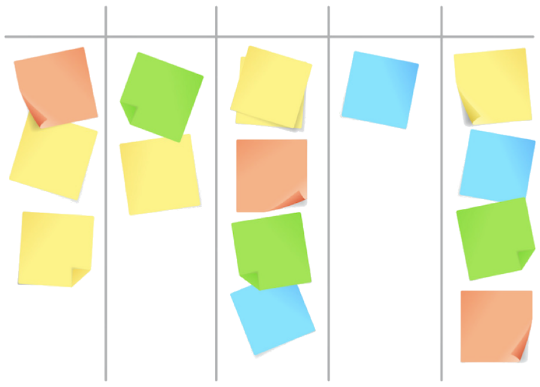


Figure 3: A planning board from the Kanban methodology [11].

2. Project planning

In this document, the reader will find documentation for all tasks that make up the project at hand. To begin, a detailed list of the tasks is shown, followed by a table that summarizes the tasks and a Gantt chart. Finally, some alternative plans will be presented in order to deal with possible setbacks.

2.1. Task definition

This section provides a list of all tasks that will need to be accomplished in order to complete the project. For each of them a brief explanation is given, followed by the resources needed to carry them out, as well as an estimation in hours about the time it will take to complete them.

2.1.1. Project management

The purpose of the tasks from this set is to define aspects like the scope of the project, the time and task planning, its budget and impact, etc, and to document them for later reference.

- **T1 Project management.**
    - **T1.1 Meetings.** One hour long meetings are held weekly, where the director of the thesis gives feedback, advice and corrections to different aspects of the project, during the duration of it. Moreover, all participants of the project are able to show their progress and share their knowledge if needed. Resources: PC and Google Meet.
    - **T1.2 Context and scope.** Write a document to give context to the project and show what the objectives are, what risks and obstacles might be encountered and what methodology is used to complete the project. Resources: PC and google services like Google Drive and Google Docs.
    - **T1.3 Project planning.** Write a document that defines all the tasks required to complete the project, a Gantt chart and possible setbacks and how to circumvent them. Resources: PC and google services like Google Drive and Google Docs.
    - **T1.4 Budget and sustainability.** Write a document that specifies the budget of the project and the sustainability measures taken. Resources: PC and google services like Google Drive and Google Docs.
    - **T1.5 Final document.** Join all previous documents to use as documentation for the project. Resources: PC and google services like Google Drive and Google Docs.

2.1.2. Project development

- **T2 Study of the GNN field (T2.1).** Perform an initial study about the concepts behind GNNs, read the articles related to different models, and identify what problems and challenges this field faces. Resources: PC.

- **T3 Study of the main programming frameworks.** Study the frameworks that will be needed to develop different aspects of the tool, like GNNs, graph analysis and the GUI.

  - **T3.1 Python Geometric framework.** Learn how to program GNNs with Pytorch Geometric, figure out how the algorithms are implemented, determine the level of customisation that can be given to a GNN without having to change the framework's very own source code. Resources: PC, GPU server, GitHub, Python Geometric,

  - **T3.2 Graph analysis framework.** Learn how to analyze graph to extract different metrics with the module NetworkX or other frameworks in the case the graphs to analyse are exceedingly large. Resources: PC, GPU server, GitHub, NetworkX, others.

  - **T3.3 GUI framework.** Learn to design and implement a simple UI with the PyQt5 or PySide2 modules to interact with the tool version of the project. Resources: PC, GPU server, GitHub, PyQt5/PySide2.

- **T4 Research and determine algorithms.** This set of tasks focuses on researching the algorithms that have the potential of improving the GNN execution the most, designing the heuristics to decide which ones to use according to a graph and designing the software structure.

  - **T4.1 Determine GNN algorithms.** Determine which GNN algorithms the tool will be able to make use of. They can be divided into these categories:
    - Analysis algorithms. Used to analyse the graph and extract the necessary metrics.
    - Preprocess algorithms. They change the representation of a graph to decrease the size in memory or improve the execution of a GNN model.
    - GNN algorithms. All techniques that can be used on a Message Passing step. Essentially, the building blocks of a GNN model. Specific techniques work better for some types of graph than others, or are faster to compute.

    Resources: PC, Pytorch Geometric, NetworkX, GPU server, GitHub, others.

    - **T4.2 Determine the heuristic algorithm.** Research what kind of algorithm that takes as input a set of metrics can work best to decide the optimal set of algorithms to use. Resources: PC, GPU server, GitHub, others.
    - **T4.3 Design the structure of the software.** Design the domain, persistence and presentation layers that will make up the tool, as well as the classes for each layer. Resources: PC.
- **T5 Implementation.** This set of tasks consists in the implementation of the concepts from T4. Resources: PC, GPU server.
    - **T5.1 GPU server and github repository set-up.** Resources: PC, GPU server, GitHub.
    - **T5.2 GNN algorithms implementation.** Implementation of the algorithms from T4.1. Resources: PC, GPU server, Pytorch Geometric, NetworkX, others.
    - **T5.3 Heuristic algorithm implementation.** Coding of the heuristic algorithm mentioned in T4.2. Resources: PC, GPU server, GitHub, others.
    - **T5.4 Software structure implementation.** Implementation of the classes and layers from T4.3. Resources: PC, GPU server, GitHub, others.
- **T6 Unit Testing.** These tasks enable a proper unit testing environment.
    - **T6.1 Unit test set-up.** Set-up a test-bed to properly test each part of the software. Resources: PC, GPU server, GitHub, others.
    - **T6.2 Unit test of each independent part.** Test each isolated part described in T4 tasks. Resources: PC, GPU server, GitHub, others.
- **T7 Tool creation.** With all implemented classes and layers, create the final versions of the software. Resources: PC, GPU server, GitHub, others.
    - **T7.1 Python tool module creation.** Create a Python module with the Persistence and Domain layer. Resources: PC, GPU server, GitHub, others.
    - **T7.1 Standalone program tool creation.** Create a standalone app by joining the Persistence, Domain and Presentation layers. Resources: PC, GPU server, GitHub, fbs module, others.
- **T8 Integration Testing.** These tasks enable a proper integration testing environment for the different versions of the tool.

    - **T8.1 Integration test set-up.** Set-up a test-bed to properly test each version of the tool. Resources: PC, GPU server, GitHub, others.

    - **T8.2 Integration test of each version of the tool.** Test each version of the tool described in T7 tasks. Resources: PC, GPU server, GitHub, others.

2.1.3. Project documentation

- **T9 Documentation.** Tasks related to the work that has to be done in order to complete the documentation for the thesis, both for the written version and the oral presentation.

    - **T9.1 Written documentation.** All the information gathered during the project is used to write the necessary documentation. Resources: PC and google services like Google Drive and Google Docs.

    - **T9.2 Oral presentation.** The contents from the written documentation is converted to a presentation form with slides and verbal explanations.

2.2. Task table and Gantt chart

2.2.1. Task Table

Macro-task name	**ID**	**duration (hours)**	**Dependencies**
Project management	T1.1	25	
	T1.2	30	
	T1.3	15	
	T1.4	15	
	T1.5	15	T1.2, T1.3, T1.4
Study of the GNN field	T2.1	50	
Study of the main programming frameworks	T3.1	30	T2.1
	T3.2	20	T2.1
	T3.3	20	
Research and determine algorithms	T4.1	50	T3.1, T3.2
	T4.2	25	T2.1
	T4.3	10	
Implementation	T5.1	5	
	T5.2	50	T4.1
	T5.3	25	T4.2
	T5.4	20	T4.3
Unit testing	T6.1	5	
	T6.2	30	T5.2, T5.3, T5.4
Tool creation	T7.1	15	T6.2
	T7.2	15	T6.2
Integration testing	T8.1	5	T7.1
	T8.2	15	T7.2
Documentation	T9.1	80	
	T9.2	40	T9.1

In total, all task durations amount to 610 hours.

2.2.2. Gantt chart

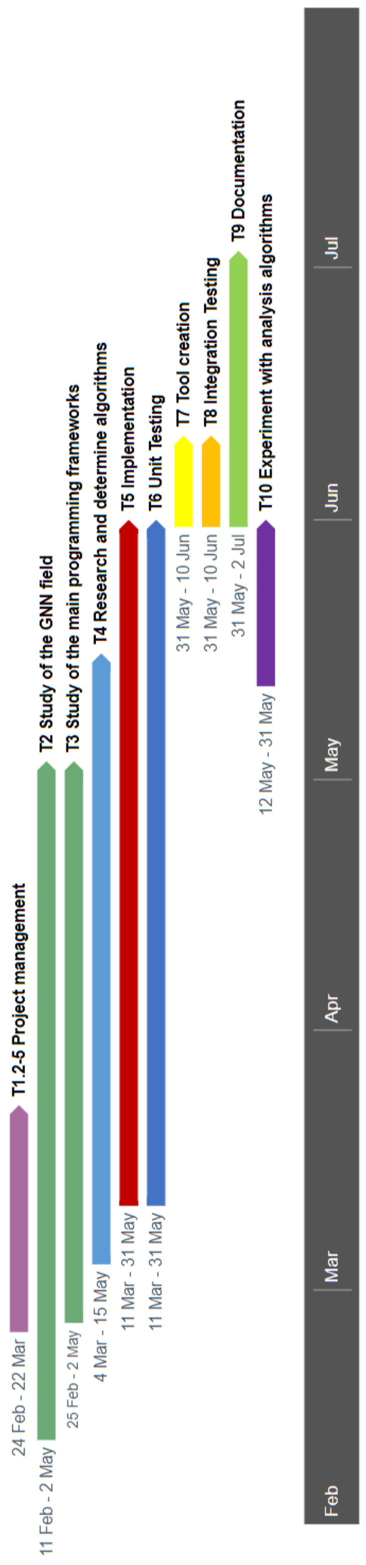


Figure 4: Gantt Chart (Own compilation)

2.3. Risk Management

During the development of the project it is almost guaranteed that some setbacks will be met. Accounting for that, here are listed some factors that could affect the planning and the scope of the project:

- **Prohibitively large graph sizes** which make a fast analysis imposible. In this case, the time spent implementing graph analysis would have to be extended a bit to explore solutions. A random subset of the nodes from the graph could be selected to perform an analysis. If the results are not good enough, some extra research might be needed to find an algorithm that properly selects a representative set of the nodes from the graph to perform an analysis. Also, using other libraries besides NetworkX, or using custom methods.
    - Tasks affected: T4.1, T4.2, T5.2, T5.3 (+15 additional hours per task).
- **Not enough costumability from Pytorch Geometric.** There might be some algorithms that improve GNN performance that are not able to work along a pytorch model. If this is the case, it will be more difficult to have this algorithm as an option. At worst, the tool could not implement it, and just let the user know that it is a good option in case she/he wants to implement a GNN with another framework.
    - Tasks affected: T4.2, T5.3 (+10 additional hours per task).
- **Debugging.** Debugging is actually the task where a programmer spends the majority of time with. The tasks related to implementation and testing have a duration that already deals with this fact in some measure.
    - Tasks affected: T6.1, T6.2, T8,1, T8.2 (+10 additional hours per task)
- **High difficulty to implement certain algorithms.** The algorithms related to GNNs and Machine Learning at large are complex. It is possible that among the selected algorithms to implement in the tool, some require an amount of time to implement that wasn't foreseen beforehand. In which case, a simpler way of implementing them might be used, although it might decrease its effectiveness. In a more extreme case, we could decide to eliminate it from the repertoire of algorithms offered.
    - Tasks affected: T4.1, T5.2 (+25 additional hours per task)

3. Background

The field of Graph Neural Networks is a relatively new field in the realm of Machine Learning. Their algorithms are specialized to deal with relational data. Relational data is a type of data where its individual components, or groups of components, have relationships of some kind with each other. Prior to its debut, relational data like network-like or graph-structured data, was first processed by some graph algorithms to extract its information, and then, other more traditional Machine Learning techniques were applied [1].

Almost 10 years ago, the field of Machine Learning experienced an explosion of popularity when techniques like neural networks started to demonstrate remarkable effectiveness in some Artificial Intelligence competitions. Neural Network techniques had been around for a long time, but the hardware limitations of the time prevented them to track attention. Models which were several layers deep, hence the name Deep Neural Networks, were able to better retain the knowledge from data.

Although the concept of GNN was conceived more than a decade ago [12][13], it has not been until a few years back that the GNN field has started to pick up steam as more and more papers are published. These articles mostly propose new types of GNNs, new algorithms and data representation techniques that make these models more effective and efficient, both in terms of time and memory. However, the field of GNNs has a set of challenges that are unique to it. This thesis will present a solution for some of them. After an introduction to the GNN field, these problems will be presented in 4.2.

To explain the concept behind GNNs, one of the papers that started this trend will be used as an example. In 2017, Thomas N. Kipf et al. published the first algorithm for a GNN in their paper "Semi-Supervised Classification with Graph Convolutional Networks". In the paper they proposed a Graph Convolutional Neural Network, or GCN, which is able to account for the relational data from a graph in order to predict the characteristics of a node, edge, or the whole network [5].

3.1. An in-depth description of GNNs

The idea from the GCN model is heavily inspired by what was a very promising kind of Neural Network at the time, Convolutional Neural Networks, or CNNs. This kind of network is specialized in image recognition tasks [5].

CNNs usually take as input a 2D matrix representing the pixels of an image or a 3D matrix, in the case of RGB color images for example, where there is a 2D slice of the image corresponding to the values of each color. CNNs take advantage of some characteristics of image data.

- Location-invariance, which means that different parts of the image contain the same patterns. No matter where you look at an image, all objects will be composed of edges and corners, for example, and what defines an edge or corner does not vary along the image.
- Importance in spatial locality. Pixels that are nearby might be part of the same thing.
- Hierarchical structure. An object in an image is composed of other smaller objects (features) and so on, all the way down to the pixel level.

To benefit from the two first principles, CNNs use filters to extract spatial data. A filter is usually a small matrix, 3x3 is a common length. Every position holds a weight that is trained in order to learn to extract some local pattern from a little place of an image, applying convolution. Since the same patterns exist all over the image. A single filter is enough for a convolution layer to search for a pattern, this way reducing the number of weights to store and train significantly. In practice, different filters tend to be used in a single layer to extract more information. Finally, if multiple layers are stacked on each other, the model will be able to recognize more abstract patterns, since it will add up the simpler ones to detect more complex information [14].

If we think of an image as a graph, it can be said that a CNN is a type of GNN. Pixels can be represented as nodes in a graph, which has a regular structure in the form of a grid.

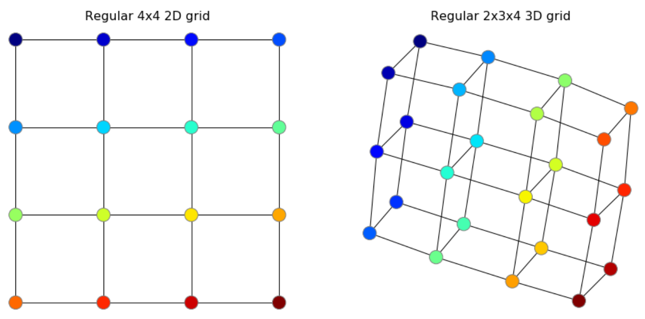


Figure 5: Regular grid graph

Each filter applies a convolution for each node in the grid graph. It takes the value of the surrounding nodes and combines them to get a new value. If we think of the next layer in a CNN as a graph, then the aforementioned process applies again. A node with the new value gets updated with information of their surrounding nodes, and so on.

Similar to CNNs with image data, GNNs can perform a series of prediction tasks. They can classify between a set of labels the nodes from a graph, or can infer the specific class of the whole graph. Also, a GNN can be designed and trained to predict the features of a node, given its relationship with other nodes from the graph. They can even predict the presence, or lack thereof, of removed nodes in incomplete data.

Thomas N. Kipf et al. proposed a model with which this process could be applied to graphs without a regular geometry, like images. Not all nodes in a graph have the same number of neighbours and, unlike images, their relative position might not be determined. There is no up and down, left or right. This means that fixed size filters can not be used.

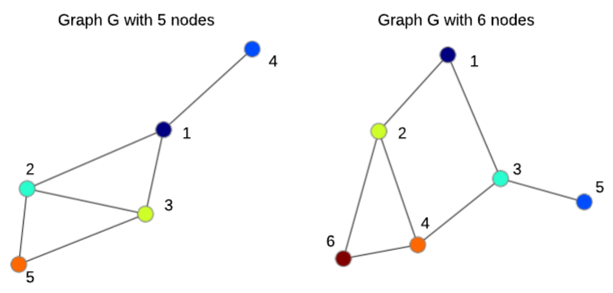


Figure 6: Irregular graphs

3.1.1. Message Passing, GNNs main operation

Two nodes from a graph are neighbours if there exists an edge that connects them. In the field of CNN, filters are responsible to extract features from the neighbours of a pixel. With GNNs, a message passing operation (or convolution) is responsible for extracting information from the neighbours of a node. It consists of reading the features of the neighbours and combining them with the features of the target node. Like CNN filters, message passing operations can be chained together in the form of layers in order to extract more information.

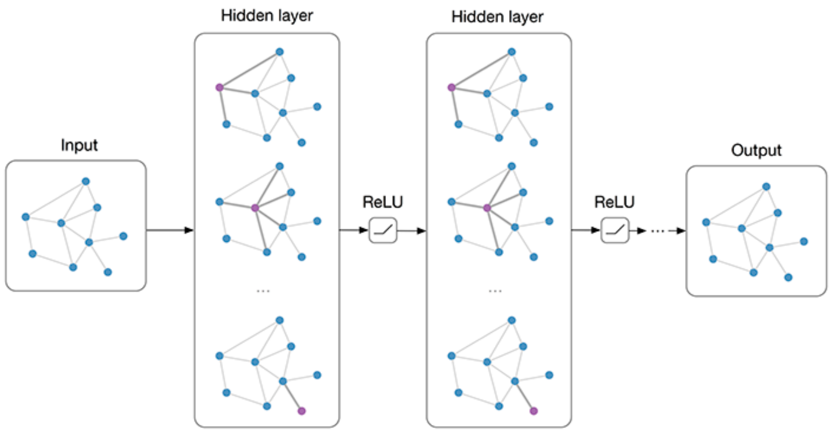


Figure 7: Layers from a GNN

As illustrated in Figure 3, message passing layers usually have the topology of their input graph, although depending on the task at hand, final layers can reduce the number of nodes or can be connected to MLPs.

The message passing operation can be mathematically expressed as follows:

$$\mathbf{x}_i^{(k)} = \gamma^{(k)}\left(\mathbf{x}_i^{(k-1)}, \square_{j\in\mathcal{N}(i)}\, \phi^{(k)}\left(\mathbf{x}_i^{(k-1)}, \mathbf{x}_j^{(k-1)}, \mathbf{e}_{j,i}\right)\right), \qquad (3)$$

where x_i corresponds to the feature vector of a node i and $x_i^{(k)}$ corresponds to the feature vector of the node i in the message passing layer k. These intermediate vectors are called embeddings. As it can be seen, a message passing layer is composed the aggregation and combination substeps:

- **Aggregation.** It corresponds to the phase where all node embeddings adjacent to the target node are aggregated. A differential function, such as a linear transformation, (ϕ) can be applied before the aggregation, which typically consists of a permutation invariant function, such as sum, mean or max ($\square$).
- **Combination.** Next, the result of aggregation is combined with the target node embedding using a differentiable function in order to compute the final node embedding of the layer k.

3.2. Problems and challenges

3.2.1. Performance

GNNs are a special case among machine learning methods. It is well known that complex machine learning methods, like Deep Neural Networks, can be computationally expensive, especially when it comes to the training phase. If the number of layers and their size are large, there are a lot of weights that have to be trained, which implies that a lot of operations have to be performed on the weight matrices. This fact is often compensated by using parallel processors like GPUs (Graphic Processing Units) or TPUs (Tensor Processing Units). The last one consists of a processor capable of performing parallel computations specifically designed and optimized for machine learning algorithms.

GNNs are no exceptions for this. Moreover, they present an additional problem. When you know the number of layers and their size, in the case of conventional Neural Networks, you can make an estimation of the time it will take to run them, since the model will always need to perform the same amount of operations. When it comes to GNNs, this is not guaranteed.

The size of the layers of a GNN depend directly from the input graph, because they share the same topology. In most cases, there exists one node in the layer for every node in the graph and one connection for every edge. This means that for the same GNN model, the time it takes to train it, or compute a prediction, can vary enormously from graph to graph [1]. Graphs are complex structures and , the time it takes for a GNN to compute a prediction on them is also dependent on different characteristics of the graph. The more edges a graph has, the more feature vectors it has to aggregate. Graphs can have different degree distributions. In one case, a few nodes could connect to a very big number of nodes, this can be the case of social networks. Another case could be that edges are more equitably distributed between nodes. These differences also play a role in the number of computations that a GNN has to perform and how effective are processors in computing them.

As the graph datasets that have to be analysed gets bigger, memory space starts to be an issue. In conventional machine learning tasks, if a dataset is too big to fit in memory, it can get split into chunks in order to load one chunk at a time, during training. This practice does not present a problem because the samples of a dataset are independent from each other. A Neural Network only cares for a particular sample or group of samples each time.

Relational data is a whole other story. Relational datasets have an additional piece of information, which is that they relate individual samples with each other using some criteria. Like in other ML fields, relational datasets are split in chunks and analysed on at a time. A message Passing operation will find in a lot of cases that some neighbours of a node are not present in the current chunk, and will have to load additional data to memory. This process is very slow and can slow down the execution of the GNN greatly.

A lot of research has been started in the last few years that aims to find solutions that can reduce the computational cost of GNNs and make a better use of memory resources. There are a great number of proposals both in the software and hardware domains [1].

Two examples of these kinds of proposals are PCGCN [15] and HAG [16]. PCGCN is a partition technique that splits the graph in dense groups of nodes. This way, there are a minimal amount of edges that depend on data that is not loaded. HAG is a technique to reconstruct a graph in such a way that those nodes that share the same results of a convolution get fused into a single node. This cuts the number of operations that have to be done in a considerable amount, since it eliminates redundant operations.

3.2.2. A large variety of GNN models

The structure of a graph and its characteristics are very diverse. There are a lot of types of graphs, and real-world networks can have vastly different properties. With this amount of diversity, research in GNNs has struggled to find an algorithm that can be applied successfully with all kinds of graphs. Instead, in recent years the number of distinct GNN strategies and techniques has exploded, each one of them trying to achieve better performance. This large number of GNN models can be disorienting when trying to decide a GNN for a particular task. Especially for those people that are newcomers to the field.

3.2.3. More layers, less performance?

The Deep Neural Network revolution that has taken place this decade has been allowed in a certain extension to the possibility of stacking a great number of layers to construct models that are able to learn complex patterns from data.

Nonetheless, GNNs seem to not be amenable to deep structures with lots of layers. GNNs generally achieve their best performance with very few layers. If more are added, then results achieve a maximum accuracy value and then start to fall very quickly. [17]

The underperformance of Deep GNNs, or DGNNs, is understood to be caused for 3 different factors [17]:

- Overfitting. The more layers, the more parameters, which means that the model has a greater capacity to memorize training data results and then fail to generalize.
- Greater difficulty to train due to vanishing gradients.
- Over-smoothing due to a great number of convolutions.

From these 3 factors, the over-smoothing phenomenon is, with a great margin, the most problematic, and it is the one that researchers in the field are trying to understand the most since it is exclusive of graphs.

When a convolution or message passing operation is performed in a node, its feature vector gets updated with information about its neighbours. After the convolution node features become more similar to those of its neighbours. If multiple convolutions are applied, nodes are able to get information from nodes that are further away from them. For example, 2 convolutions can connect the information of nodes that are level 2 neighbours, which means they are located 2 edges away from each other. This can be a great thing if the nature of a node depends on its further neighbours. However, given an enough number of convolutions, nodes in a graph have been exposed so much to the features of their neighbours that their features can end up being almost identical. Features get averaged out between all nodes and all local information that could have been useful disappear, making it impossible to distinguish classes of nodes or predict features, this phenomenon is called over-smoothing.

Luckly, researchers have already come up with some strategies that fight this tendency. By restricting the effects of over-smoothing, they have augmented the representative power of their GNNs, which have led to improved performance in specific tasks. Next, some of these approaches will be presented.

The GCNII layer [18]

Models built with this layer can perform well with up to 64 layers. In order to avoid over-smoothing, GCNII layers introduce two elements to the well known GCN equation:

$$\mathbf{H}^{(\ell+1)}=\sigma\left(\left((1-\alpha_\ell)\tilde{\mathbf{P}}\mathbf{H}^{(\ell)}+\alpha_\ell\mathbf{H}^{(0)}\right)\left((1-\beta_\ell)\mathbf{I}_n+\beta_\ell\mathbf{W}^{(\ell)}\right)\right) \tag{4}$$

where $H^{(l)}$ is the node embedding matrix of the layer l, which contains all the node embedding vectors. $H^{(0)}$ is the embedding vector before the first layer, it can be the result of applying a linear transformation to the node feature vector. P corresponds to the renormalized graph convolution matrix, which gives more weight to those nodes that have higher degree when it is multiplied H [18][5]:

This message passing equation describes how a node embedding gets updated. It depends on 2 factors.

$$\left((1-\alpha_\ell)\tilde{\mathbf{P}}\mathbf{H}^{(\ell)}+\alpha_\ell\mathbf{H}^{(0)}\right) \tag{5}$$

- **Initial Residual connection.** The paper proposes to use a residual connection on every layer. It consists of connecting the embeddings of a layer ($H^{(l)}$) with a fraction of the embeddings as they were before the first convolution ($H^{(0)}$). This way, the final representation will retain node information that was present at the beginning of the forward pass. The importance that is given to the first embedding can be set with the α parameter.

$$\left((1-\beta_\ell)\mathbf{I}_n+\beta_\ell\mathbf{W}^{(\ell)}\right)\right) \tag{6}$$

- **Identity mapping.** This part of the equations allows us to limit the importance that the linear transformation given by the trainable weight matrix $W^{(l)}$ has over the feature aggregation. Its effect can also be limited by a parameter, in this case β.

Normalization layers

Normalization layers are layers that can delay the appearance of over-smoothing by applying a linear transformation to node embeddings between message passing layers. One of their strengths reside in the fact that they are not bound to a specific layer architecture but they can be used in different GNN models.

These studies [17][19][20] show that normalization layers delay over-smoothing, although in a lot of current benchmarks these do not translate to better performance. That is due to the fact that for the size and characteristics of those benchmarks, shallow GNNs are enough to capture the relevant information.

Nonetheless, when DGNNs with Normalization layers are used in very large graphs they are able to outperform conventional GNNs, specially in node property prediction and graph property prediction. This is due to the fact that the more layers the model has, a node can receive relevant information from higher level neighbours, which results in more accurate predictions[17][19].

3.2.4. Heterophilic graphs

Among the numerous properties that a graph can have, there is the concept of homophily, or the tendency of nodes to bond with nodes of similar characteristics. It is difficult for a big graph to be perfectly homophilic, since nodes of different kinds will end up bonding some of the time. This is why homophily can be computed as a ratio for every graph. It is usually expressed as a coefficient between 0 and 1. The opposite concept to homophily is heterophily, which will be present in those graphs the nodes from which like to bond with nodes of different classes. When the homophily ratio is low, we can talk about a heterophilic graph.

GNNs are especially good with graphs that represent high homophily. Message passing convolutions aggregate the embeddings of neighboring nodes. This way, the target node embedding will resemble that of their neighbours.

As it can already be suspected, this approach does not play well with heterophily. In some cases, even non-relational algorithms can outperform a GNN struggling to generalize an heterophilic graph [21]. There are a few reasons why this is the case [22]:

- Since message passing aggregators lose the structural information of higher level neighbours, information can arrive from nodes that are not immediate neighbors. However, it might be important for a classification task to know if the information received comes from the direct neighbour or from somewhere else, but this is not possible.
- Message passing aggregators are not capable of fully capturing dependencies between distant nodes. If the features of a node depend on nodes that are distant, the useful information of this dependency can be masked by lots of irrelevant information. Also too many convolutions might be needed to pass the information, in which case overfitting might ruin the process anyway.

Like in the case of the overfitting problem, there has been some research tackling this problem too. Some solutions have been defined, two of them are explained below:

Geom-GCN [22]

The Geometric GCN model can directly aggregate feature embeddings that are similar to the target node even if it is from nodes that are far away. To achieve this message passing formula that goes beyond the neighbourhood, this model uses a different aggregation scheme in addition to the conventional aggregation scheme where neighbourhood embeddings get aggregated.

This new scheme is called a geometric aggregation scheme. It requires a new representation of the graph. This new representation consists of mapping all nodes to an euclidean space with a concrete number of dimensions *d*. A mapping function takes as input the feature embedding of a node and outputs a new vector with *d* elements, which corresponds to the coordinates of a point in the euclidean space. Every node gets to have a point in this space. Nodes with similar features will be closer together in the space. Then, edges can be added between these points. Points that are closer than a specific distance receive an edge that connects them.

Using this technique we are able to take nodes that are similar but that might be far apart in an heterophilic graph and connect them directly. Now nodes can receive information from their neighbors and also from other nodes that are similar to them but are further apart in the original graph.

H2GCN [23]

The H2GCN layer applies 3 strategies to work with heterophilic graphs:

- **Ego and neighbor embedding separation**. Nodes do not include self loops. This means that the embedding of a node does not factor in the aggregation of the neighbourhood. Instead, the embedding of the target node gets concatenated to the result of the aggregation operation in the combine operation.
- **Higher order neighbourhoods.** In the aggregation phase, not only the direct neighbours are explicitly aggregated but also the level 2 neighbours, or even higher neighbours if wanted.
- **Combination of intermediate representations.** The final embedding corresponds to the concatenation of the hidden embeddings from all the intermediate layers. This way, the final representation captures local data from the first convolutions, as well as more global data given by the last convolutions.

4. State of the art in tools and frameworks

4.1. Introduction

Almost in all cases, Machine Learning algorithms need to compute a large amount of calculations with their input data. Those calculations have to be done several times in order for the algorithm to successfully retain the knowledge inferred from the data using backpropagation techniques. Moreover, usually the amount of input data, extracted from a dataset of a specific kind, is also considerably big.

These characteristics make machine learning algorithms very costly. That is the reason why it is of great interest for all those researchers, programmers and enterprises who use machine learning to make all processes as efficient as possible. Luckily, the computation of a single step (the update step in GNNs or a convolution operation in a convolutional layer in CNNs for example) can be done in parallel since the data manipulated in said step is largely independent of each other. Parallelism can be achieved using either a Graphics processing Unit (GPU) or a Tensor Processing Unit (TPU).

Moreover, fast performing code has to be written in a compiled language close to machine code, like C/C++ or Rust. These languages are harder to use in general. Managing library dependencies, writing code, memory management, among other things, are all more complex tasks than other languages that are meant to be interpreted, like Python. In order to use the processing power of a GPU or a TPU with a machine learning program one has to also communicate correctly with the proprietary libraries and drivers that enable the use of those parallel processors.

If we were to develop and experiment complex GNN algorithms from scratch, just the process of writing under the hood code, setting it all up to run correctly and debugging the project, would burn up all the resources that should go into meaningful research, both in time and personnel, which needs to incorporate expert programers familiarized with high performance code but might know little about GNNs for example.

For all the reasons above, when a field in computer science gets popular and matures, there are a lot of frameworks and tools that appear in parallel with this growth. These tools are meant to take care of all those details that are not directly correlated to the field's knowledge but are necessary to run everything fast and efficiently. Then, all the programs, drivers and dependencies needed are packed and presented with a level of abstraction that will be usable for researchers and will enable fast prototyping and development.

These tools and frameworks use a different level of abstraction depending on their target users. For example, some of them are presented as Python modules to program algorithms in precise ways and others are able to run whole complex Neural Networks without having to write any line of code.

4.2. GNN frameworks

In the GNN domain, some programming frameworks have appeared in the last few years. Some of them are built on top of existing ML libraries, making it possible to utilize their already existing functionality while providing new dedicated kernels for GNN specific operations like Message Passing [24]. Examples of these frameworks are Pytorch Geometric or TensorFlow for geometric deep learning. The Deep Graph Library, or DGL, is a remarkable example that is built on top of multiple ML libraries like PyTorch and TensorFlow. It contains a variety of GNN implementation examples and has different kernels to perform matrix multiplication, which are used depending on the sparsity of the adjacency matrices. They claim to be up to 10 times faster than PyTorch Geometric when training [1]. However, the technicality and the quantity of their algorithms is a barrier for newcomers to the field of GNNs since they offer little help to decide what algorithms are the best among all the available.

4.3. iGNNition

iGNNition is a framework that takes a different approach to develop GNNs. It has been developed by the BNN-UPC research group. With this framework, GNNs can be designed without having to write a single line of code by using a high level description of the model. This allows for fast prototyping and deployment of GNNs without having the knowledge required to program them [25]. Some of its design principles have served as inspiration for the development of iGNNspector, Like the use of the yaml format to describe GNN models, which is a simple and easy to use yet powerful markup language. It is internally powered by TensorFlow, the models that it generates from a GNN description are implemented in this ML framework.

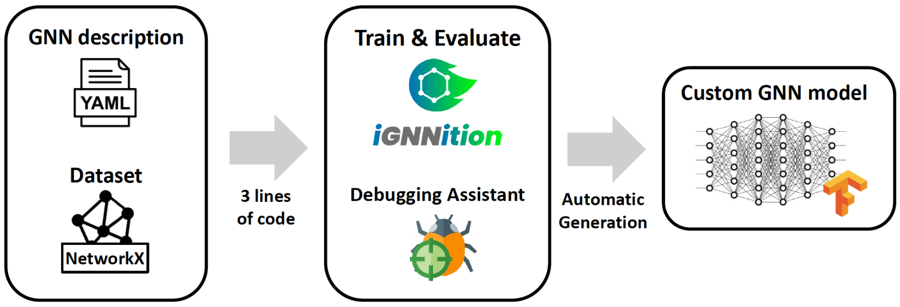


Figure 8: iGNNition workflow diagram [25].

4.4. PyTorch Geometric

Figure 9: PyTorch geometric logo

In this section, we describe the PyTorch Geometric framework in more detail, since it has been used to develop part of the iGNNspector tool.

PyTorch Geometric, or PyG, is a geometric machine learning framework built on top of PyTorch, one of the most popular machine learning libraries for Python and in general.

PyTorch is a very complete, open source machine learning framework that can be used both for researching as well as production purposes. This framework provides the user with numerous abstract entities and interfaces useful to machine learning experts, which under the hood, are implemented in fast performing machine code and are compatible to load to GPU memory and be used for parallel processing. After the developing phase, trained models can be serialized and deployed with relative ease with the tools and APIs it provides.

Between the python data structures, functions and models defined in PyTorch, one can find tensors, functions to transform data structures, like activation functions, loss functions, performance metrics, and much more, machine learning classes which go from linear regressors, support vector machines, etc, all the way to neural networks layers like simple MLPs and more advanced models like CNNs. All of them can be combined and fine tuned to develop complex models.

4.4.1. PyTorch Geometric design principles

PyG utilizes all the functionality of PyTorch to define a framework specifically designed to be used for Geometric Machine Learning, a branch of machine learning which works with relational data, where pieces of information are related and depend in some form with each other. Of course, information represented as graphs fall into this category.

The Python module is divided into 6 sections, the most important of which for this thesis are the following.

Data section

It contains the classes used to prepare and hold graph-like information. Some of the most useful are:

- **torch_geometric.data.Data.** It is the most straightforward class. It is used to represent a single Graph. Its main attributes are:
    - **x**. The node feature matrix. It contains the feature vector of every node. Its shape corresponds to [num. nodes, num. node features], which means that it holds as many rows as there are nodes in the graph, with a row length equal to the number of features of a node. Every row belongs to the node with an index equal to its position in the matrix. Simply, row 0 belongs to the node with index 0, row 1 to the node 1 and so on.
    - **edge_index.** Adjacency matrix in COO format. With shape [2, num. edges]. An edge is represented as the connection between a node index in the first row to the node index in the same position but in the second row.
    - **edge_attr**. The edge feature matrix with shape [num. edges, edge features]. Very similar to x, but with edge features.
    - **y**. The target value to train a model against. It can have an arbitrary shape. For example, if the task at hand is node binary classification, then it would have shape [num. nodes, 1], if it is node feature prediction, it might have more columns. If the task is graph classification, then it would have shape [1, *].

  To make the class more flexible, any other type of attribute can be added.

- **torch_geometric.data.Dataset.** This class allows users to load a dataset from the internet, apply a transformation to the data, add some criteria to filter the data and then save it to a specified path location. There are graph datasets that contain millions of nodes and edges. Such enormous networks will likely not fit in memory. This is why Dataset will just load and store data without loading it entirely in memory. Its main attributes are:
    - **root.** The directory where you want to save the dataset.
    - **pre_trasform.** a callable object that, if passed, will take the data (converted to a Data class), apply its transformation, and return it.
    - **pre_filter.** Like pre_transform, a callable that will take a Data object and return a boolean indicating if it should be stored or ignored.

- **torch_geometric.data.InMemoryDataset.** This class inherits from Dataset and can be used to load small datasets that do fit on memory. This way, the process from downloading, applying transformations, filtering to running a model with the data is much faster.

Datasets section

This section contains different classes that inherit from the class Datasets. Each one of them can be used to download commonly used benchmark datasets of different types. The important ones for this thesis are:

- **Planetoid.** Downloads the citation networks Cora, CiteSeer and PubMed. These graphs represent scientific papers as nodes, and citations among them as edges. They are used to predict the scientific field of the papers, since edges between papers of the same field are more likely.
- **Wikipedia Network.** Downloads to versions named “Chameleon” and “Squirrel”. The graphs consist of nodes representing Wikipedia pages and edges are hyperlinks between them.
- **WikiCS.** Downloads a similar graph than the WikipediaNetwork one.
- **Actor.** A graph where nodes represent actors and ages represent common mentions between their Wikipedia pages.
- **Amazon.** Downloads either the “Computers” or the “Photo” network. Nodes represent amazon products and edges represent the fact that 2 goods are usually bought together.
- **CitationFull.** Downloads a more large and complete version of the datasets from Planetoid.
- **Flickr.** A graph that contains descriptions and common properties of images.

nn section

This section contains classes representing a wide range of GNN layers, inspired by their respective scientific papers. Their mother class is MessagePassing. As the name implies, this class represents the common operation in GNNs of message passing, described as:

$$\mathbf{x}_i^{(k)} = \gamma^{(k)}\left(\mathbf{x}_i^{(k-1)}, \square_{j \in \mathcal{N}(i)}\, \phi^{(k)}\left(\mathbf{x}_i^{(k-1)}, \mathbf{x}_j^{(k-1)}, \mathbf{e}_{j,i}\right)\right), \qquad (1)$$

Inheriting from this class, one can define the functions message() and update() which correspond to ϕ and γ functions respectively. Internally, PyG utilizes efficient GPU scatter and gather algorithms to accelerate sparse matrix multiplication, using the edge indexes from a data object.

Since this thesis has no need for a custom GNN layer, we proceed to explain some of the layers available in the PyG module, which have been used in the GNN model aspect of the iGNNspector.

The way these layers are meant to be used is “inside” a machine learning model that represents a GNN. This class has to inherit torch.nn.Module, a class from the PyTorch library. The next code snippet shows a simple GCN model inspired by an example in the PyG documentation page.

```
class Net(torch.nn.Module):
    def __init__(self, in_features, out_features):
        super(Net, self).__init__()
        self.conv1 = GCNConv(in_features, in_features)
        self.conv2 = GCNConv(in_features, out_features)

    def forward(self, data):
        x, edge_index = data.x, data.edge_index

        x = self.conv1(x, edge_index)
        x = F.relu(x)
        x = F.dropout(x, training=self.training)
        x = self.conv2(x, edge_index)

        return F.log_softmax(x, dim=1)
```

As it can be seen in the “__init__” method, two GCN layers are initialized. The majority of layers need to be initialized with the parameters “in_features” and “out_features”, which correspond to the length of the input vector and the length that the output vector will have. Usually, for the first layer “in_features” correspond to the number of features from node feature vectors. The final layer could have “out_features” equal to the number of classes that we want to classify.

When we want to run our model, both in training mode or inference mode once trained, the PyTorch “forward” method will be called with the parameter “data” representing the graph. The majority of the time, a GNN layer is called with the feature vector (“x”) and the edge index as parameters. This way it can perform a message passing operation from the source node to its neighbours, and then updating its feature vectors.

Some of the implemented layers are:

- **GCNConv.** The simple GCN convolution. Its only mandatory parameters are “in_features” and “out_features”, although as all other layers it has more parameters, those two are enough to perform the convolution operation on the given data.

- **GATConv.** The GAT convolution layer. A parameter to specify the number of attention heads, "head" can be passed. If not, it is equal to 1. If the user wants to average the results of the attention heads instead of being concatenated, then it has to set the parameter "concat" to "False". Default is "True".

- **GINConv.** The layer from the GIN model works a little bit differently. The length of the input feature vector and the output vector does not have to be provided to it. Instead, The only mandatory parameter has to be a neural network. This neural network, in turn, has to be of a specific size. The first input layer has to have the same length as the input feature vector, and the last layer has to have the length of the output vector.

- **GCN2Conv**. This layer is designed to construct Deep GNNs as it incorporates residual connections and identity mapping to reduce the effect of over-smoothing. The input and output vectors of this layer are the same length so there is only one parameter to specify it. Additionally, an "alpha" and "theta" parameters can be specified to give more or less weight to the residual connections or the identity mapping respectively.

transform section

This section contains transformations that can be applied to data objects. One simple example can be the "ToUndirected" transform, which takes a data instance and for every edge of the graph adds another edge of opposite direction to transform it into an undirected graph.

utils section

This section contains a variety of functions to carry out very different tasks. There are a handful of functions that are particularly important for the graph class in iGNNspector. "to_networkx" and "from_networkx", as the name implies, add compatibility to use graphs previously defined with a NetworkX class. "homophily" or "degree" are examples of functions that compute a specific metric. Even subgraphs can be made with "subgraph" and others.

4.5. NetworkX

Figure 10: NetworkX logo

NetworkX, although it is not a GNN framework, it has also played an important role in the development of the iGNNspector. For this reason, this section provides a detailed introduction to it.

NetworkX consists of a rich Python library that contains numerous data structures and functions that can be used to represent and manipulate graphs, as well as to run several algorithms with them. Although it is not a machine learning framework, it is very useful when it comes to the use and study of graphs, the main pillar of GNNs.

NetworkX provides graph algorithms widely used to extract knowledge from real- world networks like social relationships, biological processes like protein-protein interaction, and human infrastructure like telecommunication and electrical lines, roads, urban centers, etc. A great portion of its algorithms are implemented in fast performing C/C++ code wrapped to be used with Python calls. It has several methods to extract graph-like data from standard and not so standard graph representation formats. That also applies the other way around. After defining a network with one of its highly customizable classes, it can be stored in the format that most suits an application.

NetworkXIt was initially developed by Aric Hagberg, Dan Schult, and Pieter Swart in 2002 and 2003 and the first official version was released in April 2005.

NetworkX provides 4 classes that represent graph data:

- Graph. Is the simpler of the 4, but it hides a lot of functionality. All edges from this kind of graph are undirected and self loops are possible. Nodes can be any kind of Python object, as long as it is hashable. You can add as many features as you want to any node via a distinct dictionary for every node. Edges can also hold any amount of features like nodes.

- DiGraph. This class inherits every functionality from Graph, but unlike the prior it can hold directed edges.

- MultiGraph. This class can hold multiple edges between the same 2 nodes, each one of them can have different features. Those edges, however, are undirected.

- MultiDigraph. Finally, this class is able to do everything the MultiGraph class can do and it can have directed edges.

5. Concept for the iGNNspector tool

As we have discussed earlier, the field of GNNs have just gained major popularity in recent years. However, it has to be noted that it is still a niche field in machine learning. Due to the relatively small volume of research conducted in the field, researchers and professionals still do not enjoy the existence of a great quantity of resources as other more consolidated fields do.

Currently, there exist too few widely used graph benchmarks. Many of them are quite small. Also, there is still not much variety between benchmarks, they are graphs from similar real-world data domains, have similar structure, and there is still a lack of datasets for some specific kinds of prediction tasks. The Open Graph Benchmark is a community-driven initiative that offers datasets, data loaders and evaluators to standardize evaluation in a unified manner [26]. They are working to add more and more diverse and great graphs based on real-world data. They also hold competitions.

Current frameworks, like PyG or GDL, although they offer powerful performance and a great level of abstraction over Machine Learning algorithms and GNNs, might still be too technical in some cases.

iGNNition is a tool that allows the user to define a GNN without using a single line of code, which can speed up the developing process considerably, but it still requires expertise both in machine learning and the concept of GNNs to create GNN models *that make sense* [25].

Those people that see the positive potential that GNNs can bring to their research or business might not have the required knowledge to develop, train and deploy their models over their data of interest. To be able to use GNN models they would have to spend time either looking for someone who has already worked with GNNs and knows how to design them, or they would have to learn the theory, inform themselves about the types of GNNs and which kind of graphs work best for them. Then they have to still learn how to use one or more of the several GNN frameworks that exist.

This inconvenience only accentuates when taking into account the particular problems that the field faces. Time and memory consumption can vary a lot depending on the structure of the graph that has to be used. Also, none of the current frameworks is able to help users decide which GNN models or techniques are best suited for their needs and the types of graphs that they want to study. These factors only make development more complex and time consuming.

This project has the intention to propose solutions that could help to tackle some of these problems by providing guidance to users about what are the best GNN options that they can deploy for their specific needs and restrictions. The chosen strategy has been to program a simple and easy to use framework, called iGNNspector, which is built on top of NetworkX and PyG and that also includes a user interface application. iGNNspector is aimed at a wide range of users regarding their level of GNN knowledge, from newcomers to more experienced professionals. This solution brings the following capabilities:

- **A multi-framework compatible graph representation.** As it has already been discussed, there are currently a number of frameworks that allow users to work with graph data and GNNs. Several of these frameworks use their own graph representation schemes and classes. Also, there exists a lot of different file formats to represent graphs, each of them with their specifications. The lack of a standard way of representing graphs, or a predominant one, makes it tedious to port graph data from une framework to another. If you want to use graph data from one place on another, you will need to have a good idea of how data is represented in both frameworks in order to correctly port the graph. iGNNspector takes care of conversions and allows users to create a graph and use it for what they want regardless of the origin.

- **Simple yet powerful analysis capability**. Knowing the structure of a graph and its characteristics is key to developing an efficient and effective GNN. Our solutions enable users to make a pretty complete analysis of the characteristics of a graph. Some graph analysis algorithms require the graph to have some particular properties, some work on directed graphs and some do not, or some might only work on connected components. Several algorithms from NetworkX and PyG have been gathered up to perform a rich analysis. The dependencies of every algorithm are dealt with under the hood so the user does not have to worry about them.

- **A simple GNN proposal system.** Users who might not know yet a lot about GNNs can learn about what models and layers exist and which ones work best for the graphs they have. Using the results from the graph analysis, iGNNspector can give to the users a set of recommendations about what layers perform best and how many of them a model has to have.

- **Code-free model building capability.** This feature is inspired by the iGNNition tool, which can generate TensorFlow GNN models. Although it is much simpler, this iGNNsector feature is able to generate PyG GNN models directly from its own recommendations, this way, users do not need to specify the model if they do not want to.

- **A different interface for users with different levels of expertise.** iGNNspector comes in the form of a programming framework for Python as well as a user interface app. The app is written in Python and is internally powered by this same framework.

6. Architecture and design

The project consists of two tools. This section presents them, explains the architecture and design behind them and provides a guide to use them. iGNNspector is open source tool and can be easily installed following the instructions from its repository hosted in the next link:
https://github.com/NilVidalRafols/iGNNspector

6.1. iGNNspector framework

The iGNNspector framework consists of a series of Python scripts which holds a set of simple to use classes and functions. The structure of the library has been thought out to be descriptive and easy to import. It has also been designed with modularity and expandability in mind. This framework has been conceived and is intended to show a particular way of how to solve some of the problems that the GNN field currently has. Given the time frame available to develop the framework, we were aware that some of the features of the framework could be expanded if we had more time. Also, we thought of additional features that could be very useful for iGNNspector that have not been able to make it to the tool. For this reason, the main classes and modules have been specifically designed to be easy to expand. If, beyond the end of this project, it is decided that iGNNspector could be expanded with these ideas or new ones, the responsible for the project could introduce new analysis methods, new graph compatibility, improved GNN model builders and so on without having to rewrite hardly any code, just add new one.

Figure 7 shows the workflow that can be followed while using iGNNspector, both in framework form and with the user interface. First, a dataset is loaded into a graph representation class. This data can come from different sources. The graph class allows us to perform an analysis of several properties of the graph, regardless of the origin of the data. After the analysis is computed, it returns a report with the resulting characterisation values and the time it took to compute them. These results can be fed to a proposer algorithm, which will propose a series of GNN models that it thinks that best suits the graph according to its properties. Users then can opt to generate a fully functioning GNN model from one or more proposals.

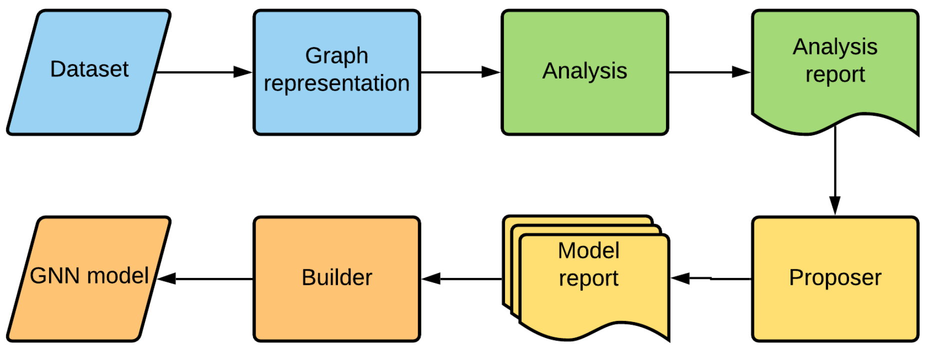


Figure 11: iGNNspector workflow

Next, we explain in more detail how to use each part. Figure 8 shows the structure of the framework library. Then we proceed to explain how every module is intended to be used.

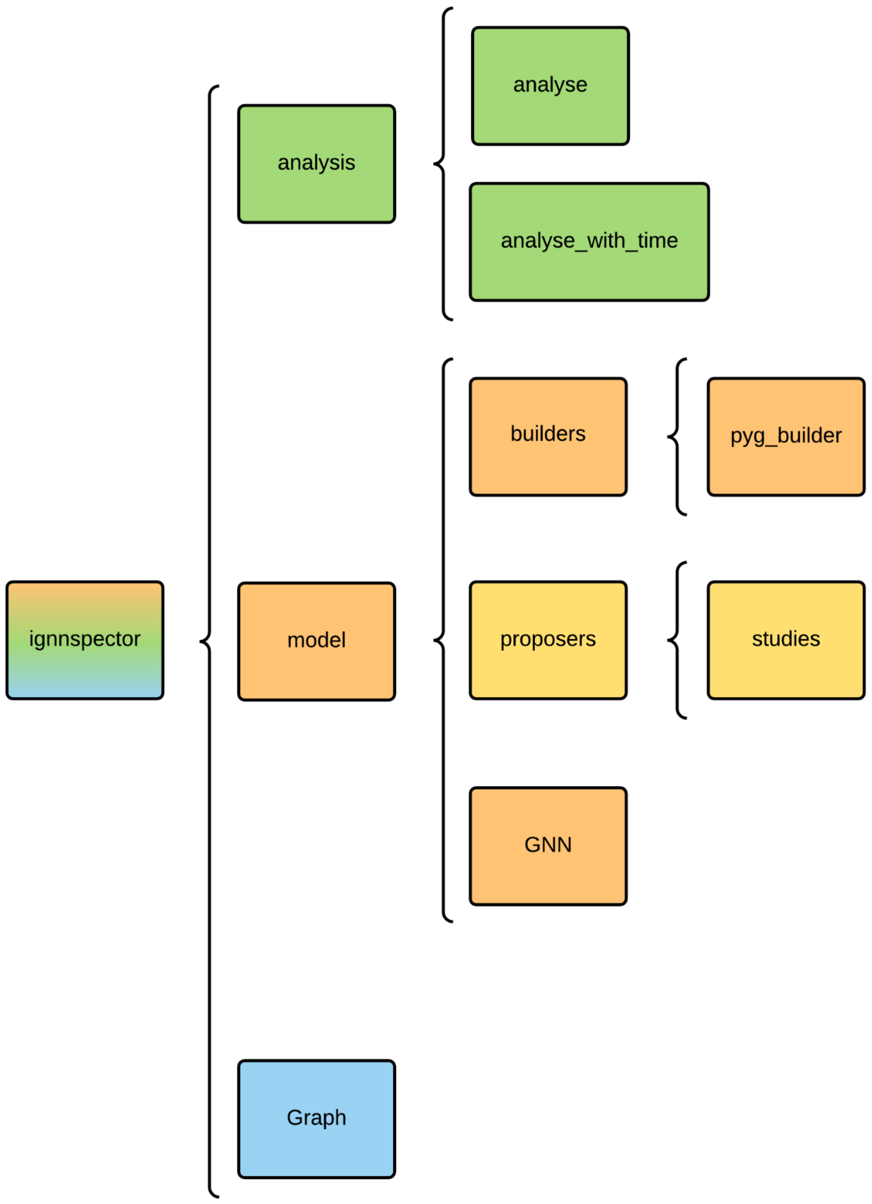


Figure 12: iGNNspector library structure

6.1.1. Graph

This is the class which represents graph data. It can be initialized with objects like a NetworkX Graph or DiGraph, a PyG Data instance or an OGB tuple. It can also receive a path to some file storing graph data of the following formats.

- GEXF, an XML format to represent graph data.
- GML. Graph Modelling Language.
- Pickle files containing an instance of the classes aforementioned.
- GraphML, another XML format to represent graph data.
- A JSON file previously generated by NetworkX.
- A YAML file previously generated by NetworkX.

Internally, a graph object can host different versions of the graph at the same time. For example, a NetworkX DiGraph instance and a PyG Data instance that represent the same graph.

If the single_representation parameter is set to True. This Graph instance will internally maintain only one version of the graph. This setting is useful when a large graph has to be loaded. Instead of accumulating different versions of the same graph, only the most recently used will prevail. This lowers the memory requirements and allows to load big graphs that take a lot of memory space. What single_representation saves in memory, however, it makes it up with time. If only one representation is used, when another representation is needed, it has to be recalculated again. If both versions exist in memory, they only have to be computed once.

Subgraphs can also be generated using either the subgraph method or the to_splits methods. Everything that can be done with the Graph class, as well as how it can be used is described below:

- **Import.** ignnspector.Graph
- **Initial parameters.**
  - **data.** At initialization receives either an object representing graph data or a path to a file representing graph data. (default: None).
  - **single_representation.** If set to True, only a single type of graph representation will exist at a time. (default: False).
- **Attributes.**
  - **num_nodes.** The number of nodes that the graph has.
  - **num_edges.** The number of edges that the graph has.
  - **directed.** True if the graph is directed. False otherwise.
  - **single_representation.** True if using single_representation mode.
- **Methods**.
  - **nx_Graph().** Returns a NetworkX Graph version of the data. If it is already builded, it just returns it, if it is not, it generates a new instance from whichever other representation is available.
  - **nx_DiGRaph().** Returns a NetworkX DiGraph version of the data. If it is already builded, it just returns it, if it is not, it generates a new instance from whichever other representation is available.
  - **PyG().** Returns a PyG Data version of the data. If it is already builded, it just returns it, if it is not, it generates a new instance from whichever other representation is available.
  - **subgraph(nodes=None, num_nodes=None).** If a subset of the nodes of the graph are given with the parameter nodes, it returns a Graph instance containing those nodes and the edges between them. If num_nodes is given, it returns a split with as many random nodes as num_nodes.
  - **to_splits(num_nodes).** It returns a generator which generates

as many random splits as possible with a number of nodes equal to num_nodes. Nodes do not repeat among splits from the same generator. The last split might contain fewer nodes than num_nodes if the total number of nodes of the graph is not multiple of num_nodes.

6.1.2. Analysis

This module contains 2 main functions that are used to analyse Graph objets. Both of them return a report of the results in the form of a map. If the user wants to save the report, it is recommended to save it as a yaml file using the python module yaml, since the yaml syntax is very readable. The metrics that are analysed are the following:

- Real average degree.
- False average degree.
- Edge cut.
- Average clustering.
- Density.
- Average shortest path length.
- Diameter.
- Radius.
- Node connectivity.
- Edge connectivity.
- Homophily. Two homophily values are displayed. Attribute assortativity coefficient uses a NetworkX algorithm wereas homophily or (homophily ratio) uses a PyG algorithm.

The report provides for each metric its resulting value as well as the time it has taken to compute it.

The analysis module contains more functions that users can use if they want. Although the main purpose of these functions is to complete subtasks from the 2 main ones, which are described below:

analyse()

This function performs a full analysis of the graph that is passed. Users can indicate the quantity of nodes to analyse, in order to reduce the time it takes to finish the analysis. The characteristics between splits can vary if they are small, this is why users can indicate if they want to analyse multiple splits and average their results. If the time parameter is provided, the analysis will try to be completed with a duration equal to the time parameter (in seconds). This function is very versatile. One or multiple parameters can be passed and the function will automatically adapt.

- **Import.** ignnspector.analysis.analyse

- **Parameters.**
    - **graph.** The Graph instance to analyse.
    - **time.** If it is not None, analyse will call analyse_with_time, explained below. (default: None).
    - **split_size.** The random number of nodes that the analysed splits have to have. (default: None).
    - **num_splits.** The amount of different splits to analyse in order to average their results. (default: None).
- **Return.** A map instance containing the value of all the available metrics and the time it took to compute them.

analyse_with_time()

This function takes a graph and the duration in seconds that the user is willing to spend analysing it. Then, the function will compute the split size necessary to perform the analysis in the expected time and will call the analyse function with this parameter.

- **Import.** ignnspector.analysis.analyse
- **Parameters.**
    - **graph.** The Graph instance to analyse.
    - **time.** The time in seconds that the user wants the analysis to last.
- **Return.** A map instance containing the value of all the available metrics and the time it took to compute them.

6.1.3. Model

It contains everything related to GNN models. With this module, users are able to get recommendations for what model characteristics are best suited for a graph, and automatically generate a PyG GNN from those recommendations without having to code. Moreover, those users with experience with PyG programming, can also opt to program a GNN by themselves using the built-in GNN class.

All of these functionalities are encapsulated in the following elements from the module.

GNN

This class represents a GNN model that can be customized. It is implemented using PyTorch and PyG elements. When initialized, it takes as its single parameter a list containing its internal components. A component represents a layer, and is formed by a map with the next 4 elements:

- **name.** A string representing the name of the layer.

- **layer.** Contains the layer convolutional corresponding to a PyG MessagePassing instance or another type of PyTorch Module instance, like Linear, which represents a linear transformation.
- **dropout.** Contains a PyTorch dropout function.
- **activation.** Contains a PyTorch activation function.

If dropout or activation wants to be skipped in a particular layer, they need to just be an identity function. When the GNN runs, components will be executed in order of appearance in the list. For each component, its elements will be executed in this order: dropout → layer → activation.

- **Import.** ignnspector.model.GNN
- **Initialization parameters.**
  - **components.** The list of components as described above.
- **Attributes.**
  - **components.** The same lists as the initialization parameter.
- **Methods.**
  - **forward(data).** It runs the GNN with the Data instance given by the data parameter. Returns the final embedding after all layers have been applied to the feature vector x.

Proposals

This module is intended to provide functions that, given an analysis report, outputs another report which contains design proposals for a GNN. Currently, there is one proposal function described below.

custom_studies()

It is a customizable proposer function. To get recommendations, the user has to give an analysis report and a feature list. The feature list is used to specify the maximum number of proposals wanted, where each proposal has a different type of feature. It has the following structure:

- [('model_type', 2), ('num_layers', 2)]

In this example we see that the first tuple indicates that the result will contain proposals of two model types. The second tuple indicates that, for every model type, there will be two proposals with a different number of layers. If we ran custom studies with these feature lists we would get at most 4 proposals. 2 models of different numbers of layers for each of the two model types proposed. At the moment only the number of model types and layers can be specified. If there were more options, they could be chained together in any specific order.

This function decides what proposals to return using a proposal tree, which indicates models based on analysis values. It uses a default tree which is written based on the knowledge gathered from GNN papers during the duration of the project.

This function is customizable because, if users want it, they can provide a custom proposal tree. It is recommended to write the tree in yaml format and then provide it to the function.

The output consists of a list where each element is a proposal. A proposal consists of a map describing the model's architecture, which can be converted to a yaml file for easy readability.

- **Import.** ignnspector.model.proposers.custom_studies
- **Parameters.**
  - **report.** The analysis report map.
  - **features.** List containing how many different types of features the proposals have to cover.
  - **proposal tree.** The proposal tree that can be passed if wanted. If it's None, the function will use the default tree. (default: None).
- **Return.** A list of proposals.

Builders

This module is aimed to provide functions that build GNN models. However, there is currently just one builder function which is described next.

pyg_builder()

This function takes as argument a proposal in the format used in custom_studies and outputs a list of components. This list follows the specification described in the GNN class section and can be used to create a GNN class instance, which will resemble the GNN model that the proposal initially described.

- **Import.** ignnspector.model.builders.pyg_builder
- **Parameters.**
  - **report.** The proposal report map.
- **Return.** A list of components to initialize a GNN class instance.

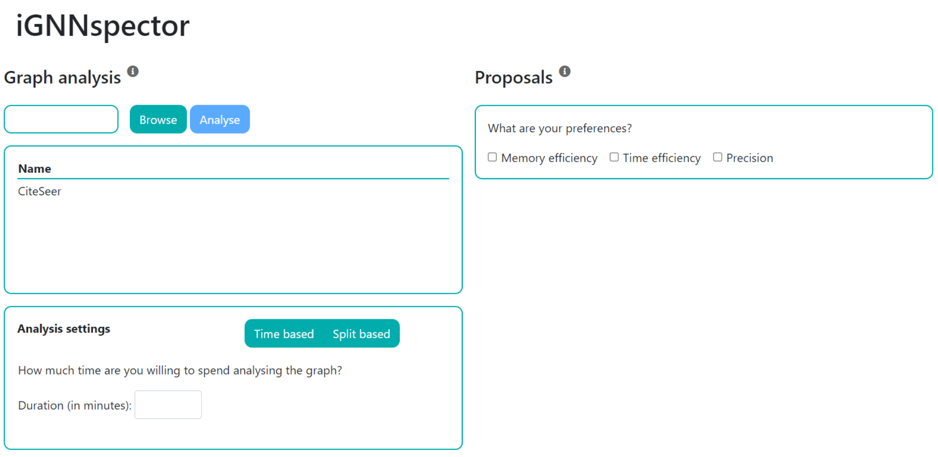
6.2. iGNNspector user interface

The iGNNspector app with user interface is designed to provide all the framework's main functionality through the use of a simple user interface. It is aimed at people that want to start using GNNs but they do not have the background knowledge about machine learning or the technical skills to program machine learning algorithms by themselves. Also it can be used as a learning tool for GNNs.

The tool provides an easy and intuitive way to load a graph, analyse it in the way users want, get proposals and finally save analysis reports, proposal reports or a ready to use GNN model.

The user interface is designed to be simple to use. Most actions that can be done with the iGNNspector framework depend on a chain of actions that have to have taken place before. For example, if we want to generate a GNN model from a proposal, we need to first complete a prior series of actions. First a graph has to be loaded, then it has to be analyzed, the analysis report has to be passed to a proposer that returns a set of proposals. Taking one of the proposals we can finally generate a GNN model with a builder.

These dependencies have been taken into account to design the workflow of the GUI app. To use the tool, users follow a linear process until they get what they want, an analysis report, proposals or a GNN model. Also they can return to any step of the workflow from any other point. The state of the GUI reminds users at all times which step they are at. In the following, an explanation of the workflow is provided with some screenshots.


iGNNspector

Graph analysis

Name

CiteSeer

Analysis settings

How much time are you willing to spend analysing the graph?

Duration (in minutes):

Proposals

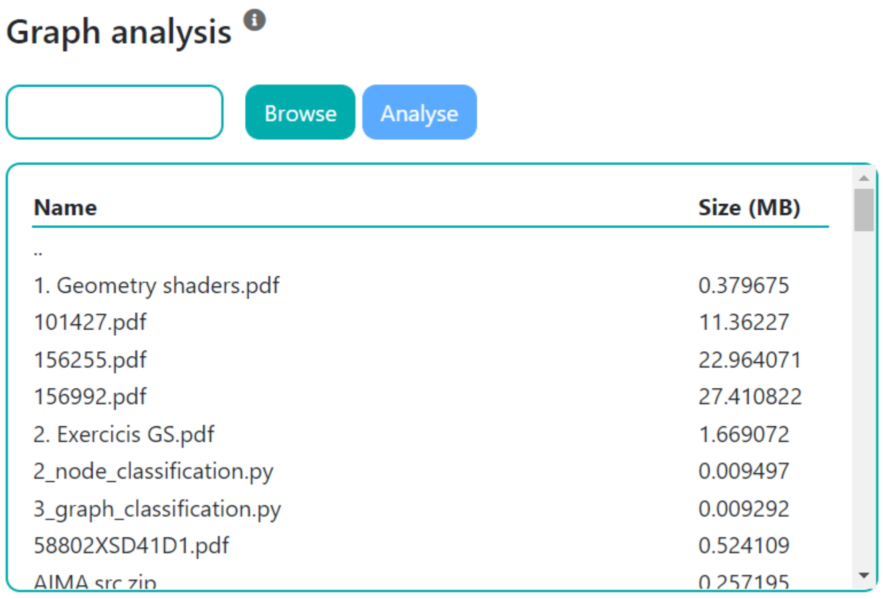
What are your preferences?

☐ Memory efficiency ☐ Time efficiency ☐ Precision

Figure 13: A complete look of the iGNNspector app at the first step.

Load a graph

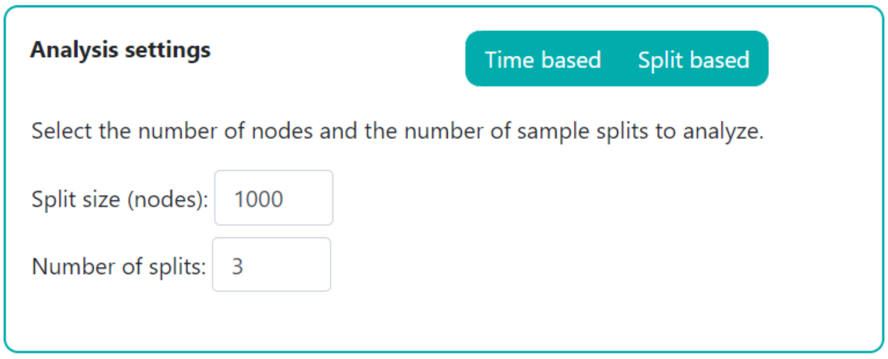
There are 2 main ways to load a graph with the iGNNspector app. The first one consists in browsing the file system and choosing a file containing graph data. The compatible files are the same as the specified in the Graph class section.



Name	Size (MB)
..	
1. Geometry shaders.pdf	0.379675
101427.pdf	11.36227
156255.pdf	22.964071
156992.pdf	27.410822
2. Exercicis GS.pdf	1.669072
2_node_classification.py	0.009497
3_graph_classification.py	0.009292
58802XSD41D1.pdf	0.524109
AIMA src zip	0.257195

Figure 14: iGNNspector app browsing window.

If the browse button is clicked a file browser bubble appears. It consists of a table with a column showing the name of files and folders and another showing the size of only files. When a file is clicked its name will appear inside the bubble next to the browse button. This means that the file has been selected.



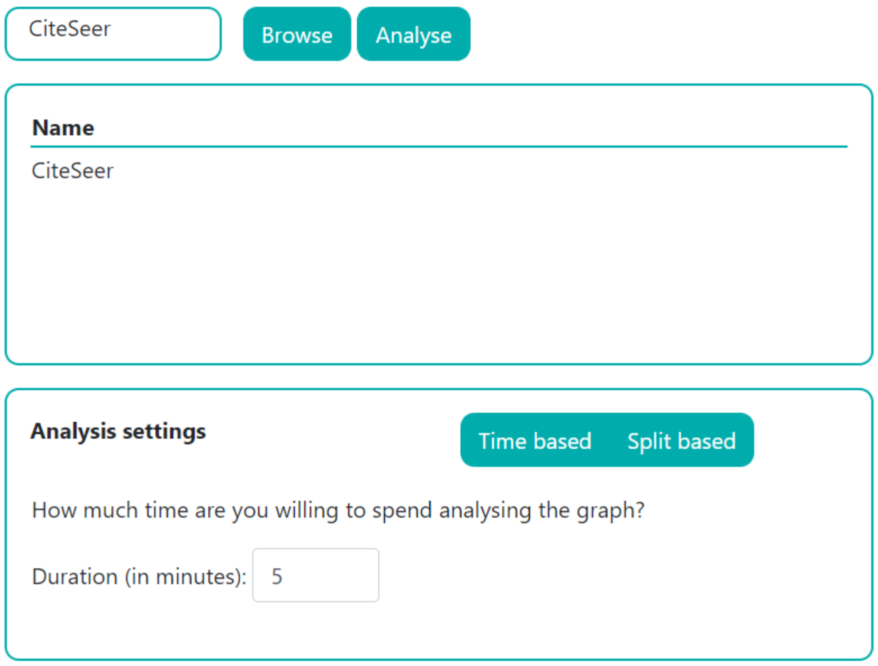
Select the number of nodes and the number of sample splits to analyze.

Split size (nodes): 1000

Number of splits: 3

Figure 15: iGNNspector analysis settings

We can configure the analysis settings the same way we would do using the iGNNspector framework. We can set no settings, one of them, or both the split size and the number of splits. If we click on the time button the app will instead ask us how many minutes we want to want for the analysis to finish.



CiteSeer

How much time are you willing to spend analysing the graph?

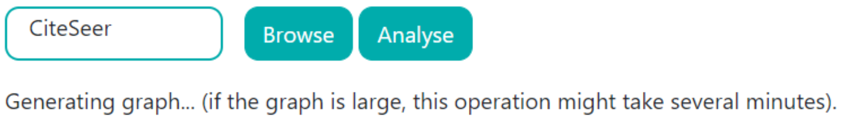
Duration (in minutes): 5

Figure 16: iGNNspector time based analysis and saved reports.

If we have previously analysed a graph, its analysis report will be saved with the name of the graph. Then, it will be listed in the table below the file browser, ready to be selected. We will be able to see the analysis results without the necessity to analyse the graph again. In Figure 8, the time based analysis settings are shown, although it will not be needed if we select a saved report.

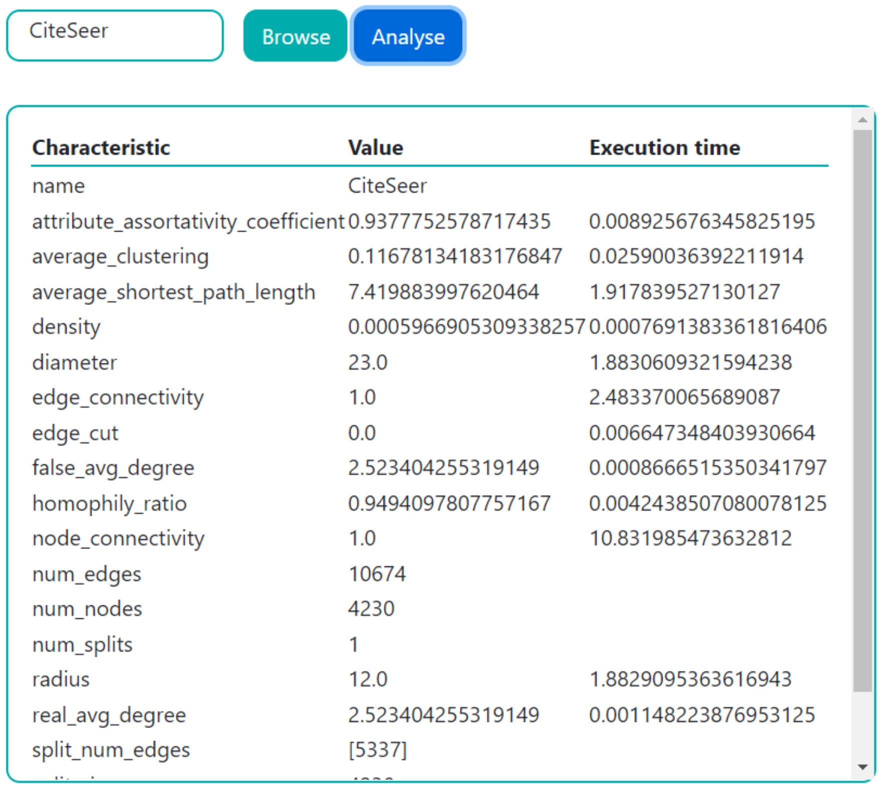
Analyse the selected graph

If a file or a saved report is selected and the analyse button is pressed, the app will proceed to compute the analysis.



Generating graph... (if the graph is large, this operation might take several minutes).

Figure 17: App graph generating message



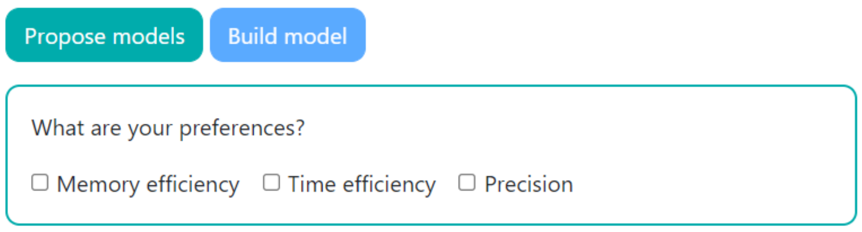
Characteristic	Value	Execution time
name	CiteSeer	
attribute_assortativity_coefficient	0.9377752578717435	0.008925676345825195
average_clustering	0.11678134183176847	0.025900363922119 14
average_shortest_path_length	7.419883997620464	1.917839527130127
density	0.0005966905309338257	0.0007691383361816406
diameter	23.0	1.8830609321594238
edge_connectivity	1.0	2.483370065689087
edge_cut	0.0	0.006647348403930664
false_avg_degree	2.523404255319149	0.0008666515350341797
homophily_ratio	0.9494097807757167	0.004243850708007812 5
node_connectivity	1.0	10.831985473632812
num_edges	10674	
num_nodes	4230	
num_splits	1	
radius	12.0	1.8829095363616943
real_avg_degree	2.523404255319149	0.0011482238769531 25
split_num_edges	[5337]	

Figure 18: iGNNspector analysis report

After the app has finished analysing the graph it will proceed to show the results in a report bubble. As it can be seen, it shows the value of a particular metric and the time it took to compute it. If we want to save the report as a yaml file, we have to click the Save analysis button.

Model proposals

Before clicking the "propose models" button, we can check our preferences for a GNN model. We can choose if we care about memory usage, execution time or good prediction results.

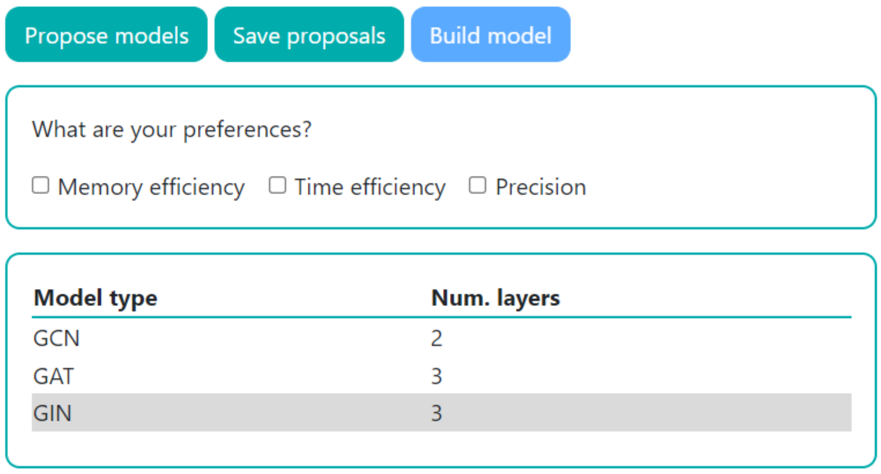


What are your preferences?

☐ Memory efficiency ☐ Time efficiency ☐ Precision

Figure 19: Proposer preferences

Once the “propose model” button is clicked, a table will appear listing the proposed models. If the “save proposals” button is clicked, every proposal will be saved as a yaml file. We can also select a proposal from the table and click the “Build model”. Then, a GNN model will be generated and saved to a pickle file, prepared to be used.



What are your preferences?

☐ Memory efficiency ☐ Time efficiency ☐ Precision

Model type	Num. layers
GCN	2
GAT	3
GIN	3

Figure 20: Model proposals

7. Implementation

This section describes how each part of the iGNNspector framework has been implemented and explains the reasons behind it. Some experiments have been performed in order to decide how a module should be better implemented, and thus they will also be explained. To get a more complete picture of the implementation we invite the reader to check the source code as a complement to the explanation that this section provides. It can be found in the following link: https://github.com/NilVidalRafols/iGNNspector

7.1. iGNNspector framework

7.1.1. Graph

The idea behind this class is to have a graph representation interface that is as agnostic as possible to the format with which the original graph data was expressed. It is meant to allow users that need to work with different graph related frameworks to load data one time and use a simple call to the Graph instance to return the particular representation needed for every framework. A good use case example for this class is the analysis module from the iGNNspector framework, which makes use of both NetworkX and PyG frameworks. Using the graph class, it has been much simpler to implement the analysis functions and its code is much easier to read and understand. This in turn has allowed for less unpleasant debugging sessions. For the moment the compatible graph objects are NetworkX DiGraph and Graph, PyG Data and a tuple representing a single graph from the OGB library.

Next, how the methods of the class have been implemented is described:

- **Initialization.** When initialising an instance of the class, the parameter data can be defined as a pathlib.Path object (which represents a path from the file system almost in the same way regardless of the operating system), or a graph object from the classes mentioned before. If it is an graph object, this function is called:

    - get_graph_from_data(data)

  It will define an attribute based on the object class. For example, if we have a Data object, self.__PyG = data. If we have a DiGraph object, self.__DiGraph = data. Also, it defines the next attributes:

    - self.num_nodes. The number of nodes from the graph.
    - self.num_edges. The number of edges from the graph.
    - self.directed. True if the graph is directed.

  It computes their value depending on the class of the parameter data. For example, to determine if the graph is directed for NetworkX objects, it suffices to check if it is a Graph or DiGraph instance. For

PyG Data objects or OGB tuples, it uses the utils.is_undirected function from PyG to determine if the edge index corresponds to a directed or undirected graph.

If data is a Path, this function will be called:

- get_data_from_path(data)

which will read the file contents according to the file suffix using a NetworkX function. If it is a pickle file it will just load using pickle. Then, get_graph_from_data will be called to perform the same actions as before.

The class contains one method for every compatible graph representation class that returns an object of that class. Those methods are:

- **nx_Graph().**
- **nx_DiGraph().**
- **PyG().**

They all work in a similar way. When they are called they check if the particular representation already exists. If it does not, they proceed to create it, assign it to a private attribute and return it.

If the “single_representation” attribute is set to True, when a method has finished creating its representation, it sets every other representation to None, in order to save memory space. This option was conceived during analysis experiments. When analysing the gbn-products dataset, from OGB, the process got killed by the system. The space needed for the original OGB data, a NetworkX Graph, Digraph and PyG representations to coexist exceeded the available memory space, which is a lot considering that the server BNN provided to develop the project has 64 GB of main memory.

- **subgraph(nodes=None, num_nodes=None).** To return a subgraph, this method needs a list with the node IDs. They can be provided with a list, with the parameter nodes. They can also be chosen at random, using the num_nodes parameter, which indicates the size of the split.

  Once we have the node IDs the next code is executed:

```
if self.directed:
    G = self.nx_DiGraph().subgraph(nodes).copy()
else:
    G = self.nx_Graph().subgraph(nodes).copy()
return Graph(G)
```

To get a split of the graph, the NetworkX representation of the graph is used to call its method subgraph() to get a view of the split. Then the method copy() is called in order to generate a true new instance representing the split. Finally the NetworkX split is used to create a new Graph instance, which will be the final return object.

- **to_splits(num_nodes).** This method returns a generator of subgraphs of the same size. It was decided to use a generator for efficiency reasons. If users dealing with a really big graph only want to get a few splits with no repeated nodes, then the generator will generate only the splits they want and no more. If to_splits returned a list instead, it would generate all the splits possible. This might take too much time. Or since there would be a whole splitted copy of the graph, the system might run out of memory.

7.1.2. Analysis

Before the implementation of this module, some experiments were carried out to seek a good way to implement analysis functions and to see how the combined computational cost of analysing different metrics evolves as a graph grows in size. First of all, we will describe and show the results of the experiments.

7.1.2.1. Experiments

Gathering data

We wanted to record the time duration of an analysis of splits of increasing size. An analysis consists of the execution of the functions shown in 7.2.1, one after the other. The settings that define an experiment are the following:

- **Dataset.** The graph used to extract the splits.
- **Initial split length.** The number of nodes of the first split analysed.
- **Node increment.** The increment in nodes with respect to the previous split length analysed. For all the final results, this value consists of a percentage. Results have been obtained with a 10% node increase with respect to the number of nodes of the previous split analysed. A percentage increment was decided to be used because, with a constant node increase, as the graph grows, there exists less and less difference with the results of one split and the next, so a lot of samples would end up being very similar. We have to also consider that as the split size grows, the cost of analysing it increases exponentially. For that, a percentage approach is the best in terms of meaningful data extraction and time efficiency.
- **Maximum splits.** With small split sizes, a graph can be divided into multiple splits. These splits might have slightly different characteristics and might take more or less time to analyse. Then, for every split size, multiple splits are analysed to save the average result. The number of splits analysed has been 5. When the size of the split grows this

number will decrease when it is impossible to divide the graph into 5 distinct splits.

The datasets used to run the experiments all come from PyG and are the following, divided by dataset loader:

Loader	**Dataset**	**# nodes**	**# edges**
CitationFull	Cora	19,793	126,842
	CiteSeer	4,230	10,674
	PubMed	19,717	88,648
	DBLP	17,716	105,734
WikiCS		11,701	297,110
WikipediaNetwork	Chameleon	2,277	36,101
	Squirrel	5,201	217,073
Amazon	Computer	13,752	491,722
	Photo	7,650	238,162

Table 1: Datasets and their node and edge count.

Motivation behind the experiments

Algorithms that compute a certain metric from a graph generally have an exponential time complexity. Nonetheless, the exact const function is unique to every metric.

With that said, we had some hypotheses.

- Although all metrics would show an exponential increase, they would grow at different rates, some growing more rapidly than others.
- The combined time complexity of all algorithms applied to splits of a graph could be predictable by a simple machine learning algorithm like Linear Regression or a Neural Network.
- If the last point holds true, maybe a single model is able to predict the time it will take to compute all metrics regardless of the graph that is being analysed.

Results

The results from the analysis experiments are shown using different types of graphics to better visualize, understand and explain them. The graphics below show the evolution of execution times from all datasets used for experimenting.

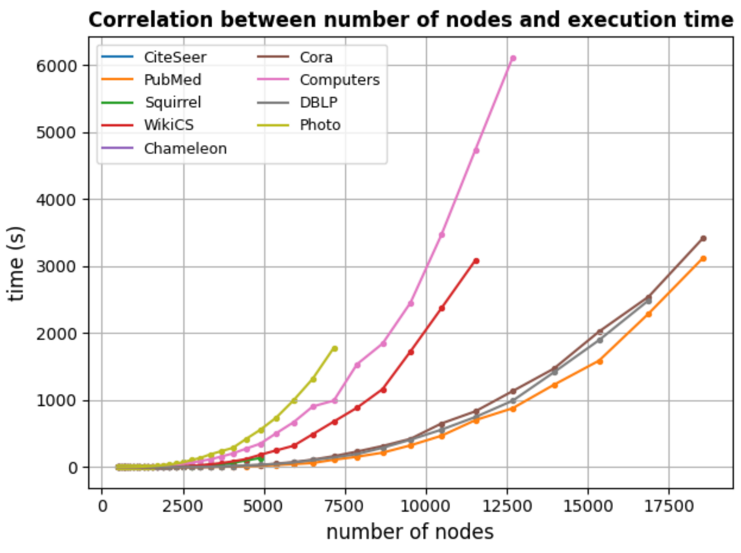


Figure 21: Correlation between number of nodes and execution time.

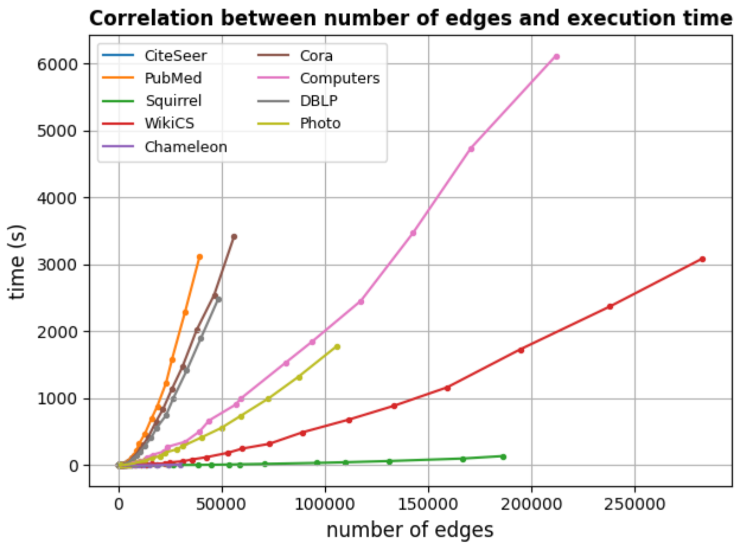


Figure 22: Correlation between number of nodes and execution time.

In figure 13, 2 groups of datasets can be seen following slightly different paths. Cora, Pubmed and DBLP grow at a lower rate. They come from the same source, CitationFull, and represent similar data. Then we have the datasets Computers and Photo and WikiCS. Their faster rate could be related to the fact that they are much more dense than the others. They have the higher number of edges from all datasets.

In figure 14 we can see 4 groups, CitationFull, Amazon (Computers and Photo), WikiCS, and Squirrel. Despite being the slowest growing datasets with nodes, CitationFull is very sensitive to an increase in edge count. The case of Squirrel is an interesting one. It has 217,073 edges, the fourth largest edge count among the datasets. However, edges contribute almost nothing to the time increase, which might be due to the fact that it is the smallest in terms of nodes, with only 2,277. With this data, the relationship between node and edge count and time execution is not wet clear.

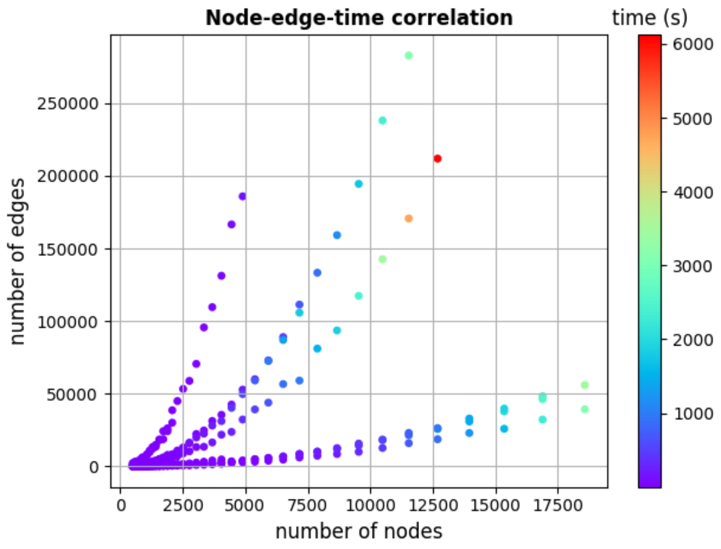


Figure 23: Correlation between the node and edge count and execution time.

Figure 15 contains a point for every sample from the experiment. The point position represents the node and edge count and its color represents the execution time of the sample. Although it is not explicitly shown, the growth of individual datasets can be traced, since points are arranged in continuous lines. There, some groups can be seen , as in figures 13 and 14. Among datasets of the same group, execution time is similar for similar node-edge coordinates. Nonetheless, there are some examples that make it evident that topology plays an important role when it comes to execution time, not just node and edge count. In the 10,000 - 12.500 node range, datasets with less edges take more time to execute than datasets with more edges.

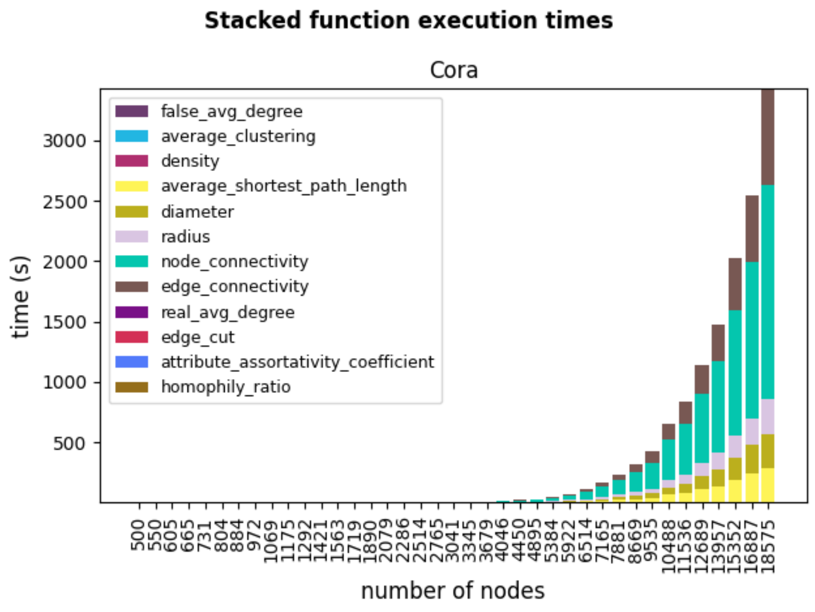


Figure 24: Individual function execution times stacked vertically, ordered by node count. From the CitationFull Cora dataset.

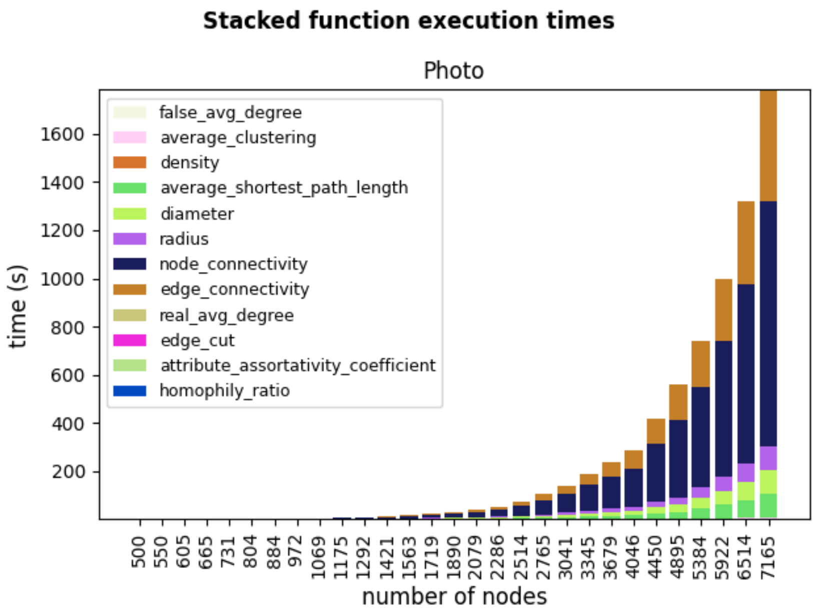


Figure 25: Individual function execution times stacked vertically, ordered by node count. From the Amazon Photo dataset.

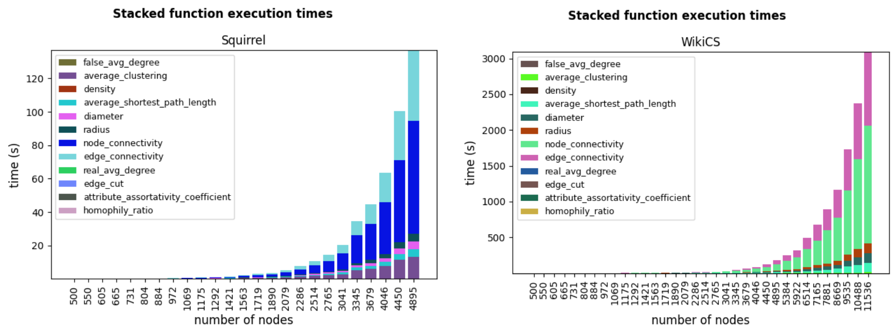


Figure 26: Individual function execution times stacked vertically, ordered by node count. From the Wikipedia Network Squirrel and WikiCS datasets.

These bar plots show the total execution time of the splits of a dataset, each bar is divided by individual function times. Looking at figures 16 and 17, it is clear that the next functions account for the majority of the execution time:

- **node connectivity.** Searches a node which, if removed, the graph would cease to be a single connected component.
- **edge connectivity.** The same concept applies to edge connectivity but with edges.
- **average shortest path length,** computes the average of all shortest paths among nodes.
- **diameter.** Finds the longest distance among the shortest paths between 2 nodes of the graph.
- **radius.** Finds the shortest distance among the distances of any node to its farthest node.

These algorithms are very costly and are very sensible to node or edge increases. They need to traverse the graph one time for every node or edge in order to make sure to find bridge nodes or edges, in the case of node and edge connectivity, and all distances between nodes for the rest.

After seeing these plots, it is clear that the number of nodes and edges are not good predictors for execution time. The next plots show the prediction times given by an MLP, which makes it clear that more metrics are needed if we want to make better time predictions.

Because MLP applies linear transformations, it would be hard for it to output an exponential result as its prediction. For this reason, the MLP predicts the base 10 logarithm of the execution time, which is a linear function. Then it is reconverted to show actual time values. Node and edge count are their input values. Although it might not seem necessary, the number of nodes and edges are also converted to base 10 logarithm because it shows better results.

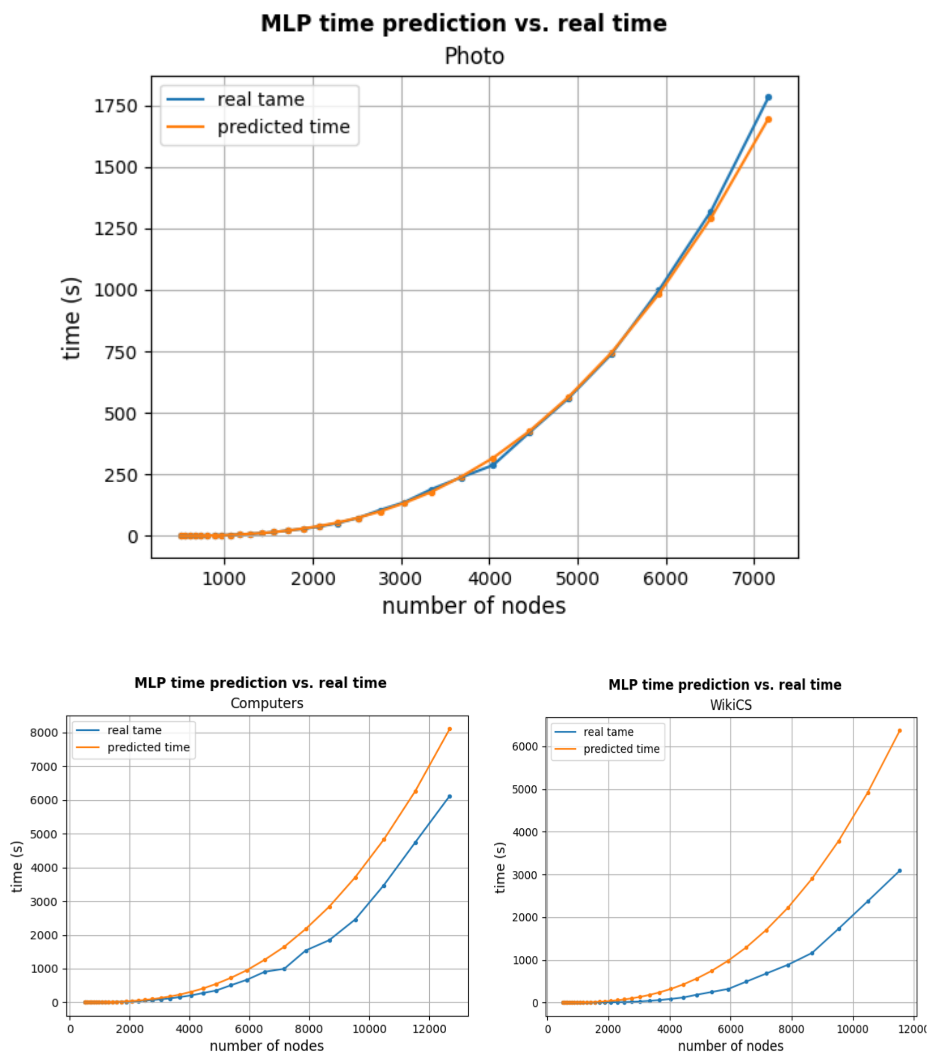


Figure 27: Predicted time vs. real time for Photo, Computers and WikiCS datasets using an MLP.

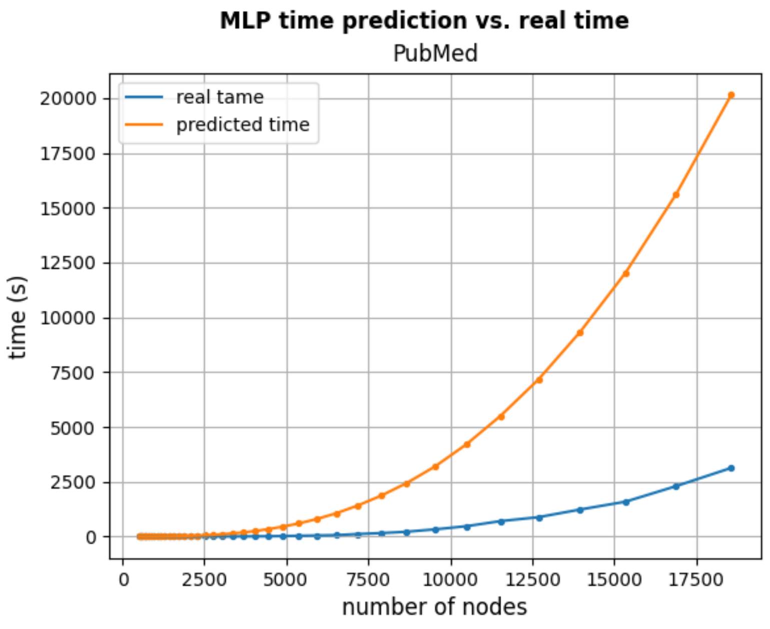


Figure 28: Predicted time vs. real time for PubMed datasets using an MLP.

The MLP model obviously fails to generalize. It makes good predictions for the Amazon group of datasets, but it is at the expense of not being able to predict the CitationFull group, which has a growth rate lower than the other datasets. Nonetheless, it does predict the execution time of some datasets with acceptable results. As it can be seen in all figures, every dataset follows a very uniform exponential function, with a constant growth rate.

This opens the possibility to use a machine learning method to learn the specific growth function of a single dataset, and then use it to inform the user how much time a dataset will take to be analysed given the size of the graph. The inverse could be done too. Let the user decide how much time they want to spend analysing a graph and then infer the size of a split.

The growth function of the execution time of a dataset is a simple one. For this reason, a Linear Regression algorithm is enough to use for single dataset prediction. The following plots show different predictions from a Linear Regression model.

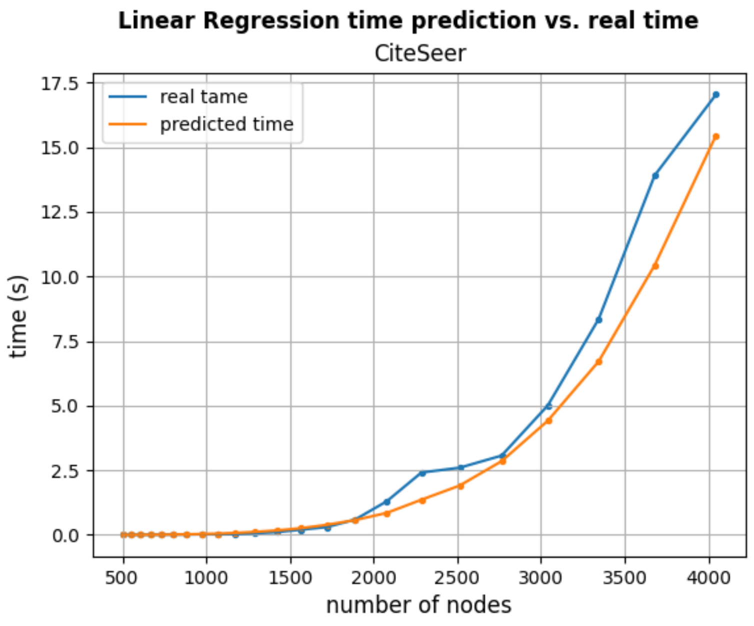
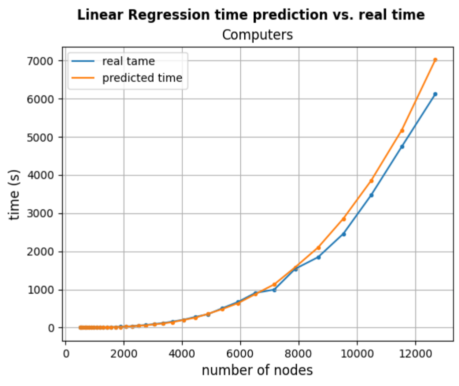


Figure 29: Predicted time vs. real time for the CiteSeer dataset using Linear Regression.



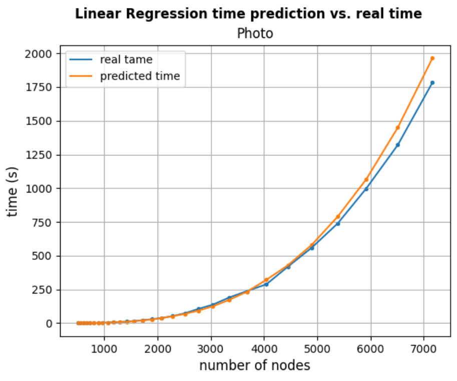
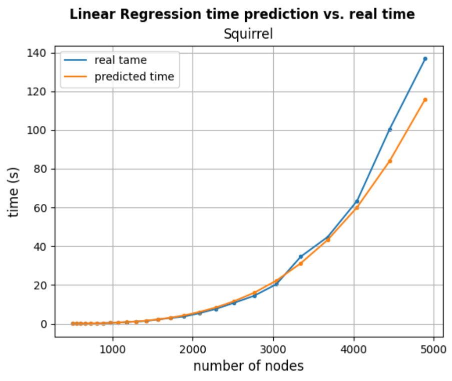


Figure 30: Predicted time vs. real time for the Amazon Computers and Photo datasets using Linear Regression.



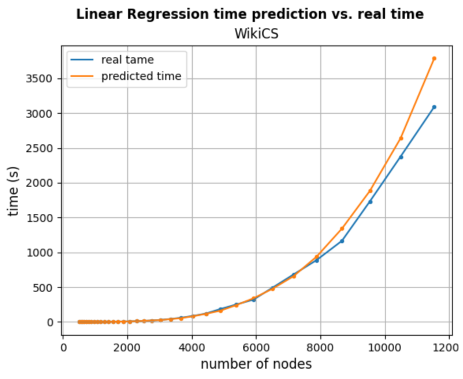
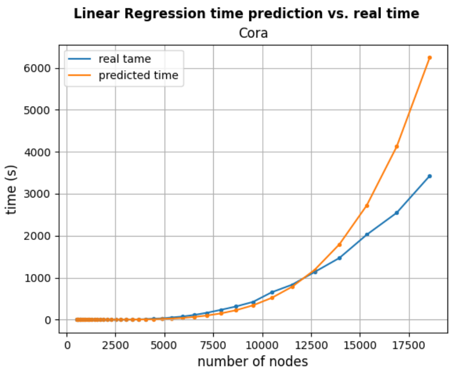


Figure 31: Predicted time vs. real time for the Squirrel and WikiCS datasets using Linear Regression.



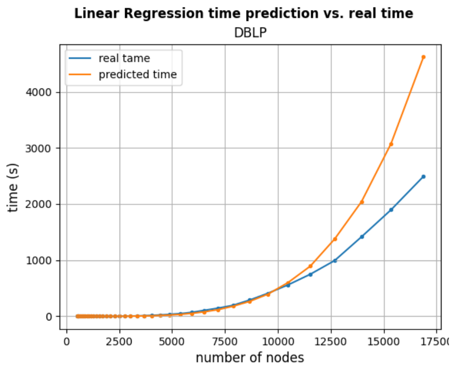


Figure 32: Predicted time vs. real time for the CitationFull Cora and DBLP datasets using Linear Regression.

The Linear Regression model seems to make accurate predictions for the majority of the datasets. However, it predicts higher times than expected for Cora and DBLP. This could be due to the fact that these 2 datasets have a very slow exponential growth. Since the algorithm predicts the logarithmic value of the time, it might be impossible to output a line with such low curvature. This shows us that in some circumstances, the model could give conservative estimations and select a smaller split size than necessary.

Conclusions

With these experiments we have observed that although different graphs increase their execution time at different rates, they do it at a steady pace, with a constant growth rate. With that and the results of the MLP, it is clear that a single model is not able to generalize the prediction of execution time given only the node and edge count. However, the constant rate with which a dataset time grows makes it possible to use a simple ML algorithm such as Linear

Regression in order to make good predictions about execution times of a single dataset.

7.1.2.2. Immplementation

Currently, 2 analysis functions are available in the iGNNspector framework. One of which makes use of the Linear Regression models used during the experiments to let users define how much they are willing to wait for analysis results.

Analysis report

Analysis functions return a map that stores the value of each metric that has been analysed as well as the time it has taken to compute it. A fragment of an analysis report is shown below converted to yaml format:

```yaml
name: 'CiteSeer'
task: node_classification
attribute_assortativity_coefficient:
  time: 0.008925676345825195
  value: 0.9377752578717435
average_clustering:
  time: 0.02590036392211914
  value: 0.11678134183176847
average_shortest_path_length:
  time: 1.917839527130127
  value: 7.419883997620464
density:
  time: 0.0007691383361816406
  value: 0.0005966905309338257
diameter:
  time: 1.8830609321594238
  value: 23.0
```

After an analysis, other entries like “name” or “task” can be added if they are useful. For example, a proposer could make use of them.

analyse(graph, time=None, split_size=None, num_splits=None)

This function can be called with any combination of its parameters. If time is given, the others are ignored and the “analyse_with_time” function is called, which will be explained later. Otherwise, the parameters “split_size” and “num_splits” are checked and corrected if either of them are greater than their maximum values or are “None”. If both of them are “None”, all the graphs are analysed. A generator is set using the “to_splits” method with the corrected values.

Then, the report map is initialized with a key for every metric available. Metric functions are stored in 3 lists located in the functions.py script, “ignnspector_functions”, “nx_funtions” and “pyg_functions”.

Once we have the generator and the report, the function proceeds to start the analysis. Every split of size “split_size” is analyzed “num_split” times and the resulting values are averaged. Once it's done, it returns the analysis report map.

analyse_with_time(graph, time)

First of all, the time is divided by 3, because 3 splits of equal size will be analysed in order to average the results. After that, the function “get_best_model” is called. This function takes 10 sample splits of a graph with increasing size, starting from 1% the number of nodes. If no edges are present with 1%, it increases the size with another 1% until it contains edges. These samples are analysed and used to train the next Linear Regression models from the sklearn library:

```
    models = [
        LinearRegression(),
        Ridge(),
        BayesianRidge(),
        LassoLars(alpha=.1),
        Lasso(alpha=0.1),
        ElasticNet()
    ]
```

After training, the model with the least mean square error is returned. Then, the model is used to decide the split size correlated with the execution time specified by the user.

Once we have the split size, the “analyse” function is called with the split size value and a number of splits of 3. It will analyse the splits and return a report map as we specified earlier in this section.

7.1.3. GNN

This class inherits from torch.nn.Module, which is the base class for all neural networks in the PyTorch framework. A PyTorch Module can represent a layer or even a whole neural network model. Module is built in a way that it can contain other Module objects, making it possible to nest them in a tree structure to create complex neural networks .

The structure of a GNN object is defined by a map of components, the structure of which is already described in “architecture and design”. When a GNN object is initialized, it runs the next code:

```
self.components = components
# a name list will be necessary to retrieve the corresponding
steps
self.names = list(map(lambda c: c['name'], components))
for name, layer in map(lambda c: (c['name'], c['layer']),
components):
    self.add_module(name, layer)
```

The components list is set as an attribute, because it will be needed later on. The PyTorch Module class is designed to configure itself depending on if it is used during training or not. For that reason, we need to add the layer inside a component as a child module of the GNN. To add a child module, we need to also provide a distinct name for it. A list of names is set as an attribute to be used later on. Then, a for loop is used to take the "name, layer" pairs from components and initialize them with "self.add_module(name, layer)".

When the "forward" method is called, it takes the node feature vector "x" and the "edge_index" from the data object. Then it proceeds to execute each GNN layer. It makes sure that layers are executed in the order that they appear in the "names" list, which is the original order that appears in the "components" list.

A for loop executes the layers. For each iteration, first it executes the dropout function, followed by the layer convolution and finally the activation function. When finished the final node embedding is returned.

7.1.4. Proposers

The module proposers, as analysis, can also be expanded by adding new scripts containing proposer functions. For the moment, it contains the script for the custom_studies proposer function.

custom_studies(report, features=None, proposal_tree=None)

Proposal tree

This proposer function takes a report map with the iGNNspector analysis format and uses the results to look up a proposal tree, from which it gets different model proposals. The proposal tree consists of a map of a specific format, and users can provide their own map with the parameter "proposal_tree". It is recommended to write the tree first in a yaml file, since it is easier and intuitive. The default map that the function uses was originally defined in a yaml file. In order to explain the format that proposal trees have, a little example is next shown written in yaml:

```yaml
homophily:
    '> 0.5':
        -   model_type: GCN
            num_layers: 2
        -   model_type: GAT
            num_layers: 3
    '<= 0.5':
        -   model_type: Geom-GCN
            num_layers: 4
        -   model_type: H2GCN
            num_layers: 4

task:
    '== node_classification':
        -   model_type: GCN
            num_layers: 2
        -   model_type: GAT
            num_layers: 3

    '== node_feature_prediction':
        -   model_type: GCN2
            num_layers: 16
```

A proposal tree has 3 types of nodes: metrics, conditions and leaves. Metrics have the same name as they appear in an analysis report. Every metric node is followed by condition nodes. In this example, the metric homophily is followed by 2 conditions, "> 0.5" and "<= 0.5". If we executed "custom_studies" with the little tree above, it would first see the homophily node. Then, for every condition it would check if the homophily value from the analysis report satisfies the condition. If it does, it would enter the condition node and find the leaves of the homophily "branch", which are a list of GNN proposals. These proposals are appended to a list.

After that, it goes to the next metric node and repeats the process until there are no branches left. Note that the "task" metric does not appear in an analysis report. This is because a proposal tree can have metrics with any name. If a user decides to add something that considers important to an analysis report, it will be used to propose models as long as there is a metric node for it in the tree.

Moreover, since this is a tree, a metric node can also be the child node of a condition node. This way nested metrics can be created. The next fragment is an example:

```yaml
memory_efficiency:
    '== low':
        task:
            '== node_classification': &node_classification
                -   model_type: GCN
                    num_layers: 2
                -   model_type: GAT
                    num_layers: 3

            '== node_feature_prediction':
                -   model_type: GCN2
                    num_layers: 16
    '== high':
        task:
            '== node_classification': *node_classification

            '== node_feature_prediction': *node_classification
```

In it we can see that for every setting of “memory_efficiency” there are different proposals for the “task” metric. Also, if we want to repeat a fragment of the tree we can use an anchor, which is a feature of yaml that allows us to “copy and paste” the contents of an entry. The contents to be copied will go after the name of the anchor, in this case “&node_classification”. Then, the contents will be pasted in the entry where we write “*node_classification”.

Proposal selection

When all the proposals have been collected, then the function proceeds to order them according to the feature list settings. With a feature list like this:

- [('model_type', 2), ('num_layers', 2)]

The contents of a proposal list could be the following:

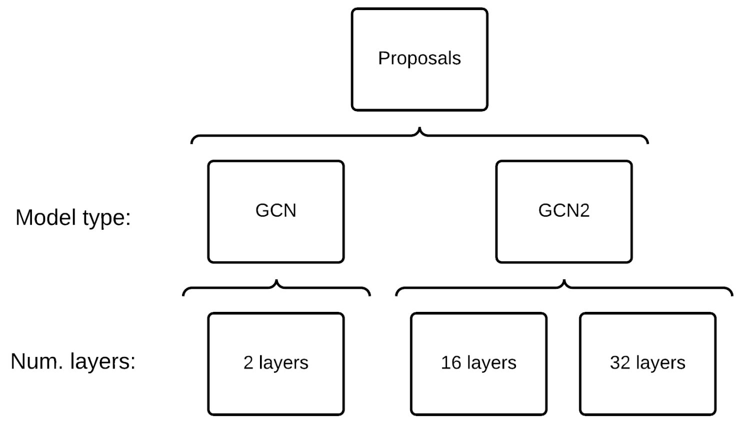


Figure 33: A proposal list structure from the “coustom_studies” function.

Although the proposal list that “custom_studies” returns has no hierarchy, this figure shows how proposals are selected internally. All of the proposals given by the tree are passed to the function:

- select_proposals(features, proposals)

This is a recursive function. Its pseudo-code can be expressed as follows:

```
def select_proposals(self, features, proposals):
    result = []
    if len(features) == 0:
        result = remove_duplicates(proposals)
        return result

    feature, num_proposals = features.pop(0)
    proposals = divide_propsals(proposals, feature)
    proposals = proposals[:num_proposals]
    for group in proposals:
        result += select_proposals(features, group)
    return result
```

- If there are no features in the feature list, then it returns all the proposals without duplicates.
- If there are, this function pops a feature from the feature list as if it were a queue.
- Then, it divides proposals into as many groups as there are different values for the feature. For example, with the feature list above, it would group all GCNs in one group, all GCN2s in another and so on, since the first feature is "model_type". Then it would take only the first 2 groups, GCN and GCN2, which are the most abundant.
- Following, it would call itself recursively for every group. This time, the most abundant 2 num. layers would be selected.
- Finally, as in Figure 33, it returns this proposal list with:
    - GCN with 2 layers,
    - GCN2 with 16 layers,
    - GCN2 with 32 layers.

Proposal report

The proposal list then is passed to the "get_report" function. It will generate a proposal report for each proposal. These reports consist in a map and will be the return value of the "custom_studies" function.

In yaml format a proposal report looks as follows:

```
layers:
- activation: relu
  in_features: 20
  out_features: 15
  type: GCN
- activation: relu
  in_features: 15
  out_features: 10
  type: GCN
- activation: log_softmax
```

```
  in_features: 10
  out_features: 5
  type: MLP
```

It defines the internal structure of the proposed GNN. “layers” contains a list with the components of each layer of the model. The “get_report” function lowers the size of the input features to match the size of the final output features (both of them are indicated on the analysis report).

7.1.5. Builders

Like the analysis and proposers module, builders can be expanded by adding scripts with functions that take as parameter a report and outputs a GNN model. The currently available builder function is the next.

pyg_builder(report)

This builder function returns the components for a GNN class object given a proposal report. For every layer on the report, “pyg_builder” builds a component map with a dropout function, the proper GNN layer and the proper activation function. Then adds all components into a list and returns it.

If the report does not mention dropout or activation, they will be identity functions in a component. An identity function returns the same value that is passed as a parameter. In Python they can be expressed as “lambda x: x”.

The current GNN layers and activation functions that can be loaded into a components list are given by the next 2 maps from the source code:

```
layer_map = {
    'GCN': get_GCNConv,
    'GAT': get_GATConv,
    'GIN': get_GINConv,
    'GCN2': get_GCN2Conv,
    'MLP': get_MLP
}

activation_map = {
    'relu': F.relu,
    'log_softmax': F.log_softmax
}
```

The keys in these maps are the ones that can be used to specify the layer and activation of a proposal report.

8. Conclusions

The work of this thesis is a response to the distinct problems that the GNN currently faces. The motivation behind the project has been to explore new ways to tackle these challenges and propose a specific solution. The iGNNspector tool can be used out of the box. It has been designed to be easily extendable, so it can also be used as the foundation of a more complex and capable framework, if in a future project it is decided to do so. Its development has also been a way of brainstorming new interesting ideas that could also be carried out on future projects.

Initially, we intended to develop a tool that was used exclusively with a user interface, since it was thought to be used by machine learning programmers and beginners in the field alike. Nonetheless, we realize that its functionality could also be useful if used separately. We thought that it would be interesting to develop it as an open programming framework that was versatile and easy to expand. This way, separate elements like the graph interface, the analysis capability, the GNN builders, etc, could be used at will. Then, the user interface could as easily be implemented on top.

While implementing the tool, some research was made about how researchers are trying to solve problems like the over-smoothing phenomenon or heterophily, in order to better understand them and think how iGNNspector could help with them. Although, given the development time frame, little connection has been done, the knowledge gathered has been included in the thesis.

The final result of the project is both an open source programming framework and a user interface app. In its current state it can already be helpful. Nonetheless, we find that its biggest value is the fact that it can help newcomers learn the concepts and particularities of the GNN field. Also, the new ideas that it has brought to the table can be interesting to explore in the future.

8.1. Further development

Further development ideas range from new kinds of proposals and builder algorithms to entirely new solution concepts.

First of all, besides the Linear Regression algorithm that powers the "analyse_with_time" function, no other ML algorithms have been used elsewhere. It would be interesting if a proposal function used some ML technique. But for that, a lot of types of GNN would need to be tested for different types of graphs and record their performance. Such activity has remained outside the possibilities of the project.

Also, a proposal could be made to output an iGNNition model description, that could then be feeded to iGNNition to generate TensorFlow based models.

Currently, the model report format that is used by the proposals and the PyG builder is fairly simple. As the tool gets compatible with more GNN models and techniques, a revision of this format would be very useful.

There exists a lot of GNN development frameworks like, TensorFlow, PyG, DGL, etc. If it is needed, there can be as many builders as different GNN frameworks.

An analysis could be much more configurable. Users could be able to specify what metrics they want to be computed, or what metrics they want to exclude.

Needless to say, the Graph class could be made compatible with more graph frameworks. Moreover, currently splits with nodes chosen at random is the only way to get splits. For some topologies, if nodes are chosen at random and the split is too small, it might contain very few edges since almost none of the nodes from the split are neighbours. To solve problems like this, new methods to split graphs could be introduced. Like an algorithm that gives an initial node, it traverses the graph to take x number of nodes. This way all nodes would be connected. Also, splits based on edges, not nodes, could be introduced.

Maybe some research could be done to determine what metrics a ML algorithm needs to properly predict analysis times regardless of the dataset. If that were possible, "analyse_with_time" would lose no time taking some samples from a dataset.

Our director also has pointed out that iGNNspector could be brought to the web for everyone to use it. This way, users could agree to share their graphs and graph analysis with us. With all the data gathered, we could be able to better understand the topology of all kinds of graphs and their relation with the performance of different types of GNNs. Also, it could contribute to the search for more and better dataset benchmarks, which is a crucial part of GNN development.

8.2. Technical competencies

With regards to the technical competencies of the project, I think they have been accomplished. The experiments carried out have allowed me to fine tune the implementation of the graph class and analysis functions in order to make them efficient. Also, the recommender system of iGNNspector reflects the research that has been done to determine the most suited GNN algorithms for a given input. Also, a range of ML algorithms have been used throughout the development, like MLPs, Linear Regression, and GNNs. Finally, the code of the iGNNspector tool has been structured in a way that makes sense, is easy to understand, easy to traverse and with modularity and expandability in mind.

8.3. Personal acknowledgement

With this project I have been able to learn a lot about a unique field of ML. The GNN field has a unique set of characteristics that separate it from other ML fields. Their algorithms work in ways that I could not have foreseen before I learned about them. For these reasons, since the first moment our director presented the concept to me, GNNs have held my interest and attention. I am happy to have worked on the development of the iGNNspector because I have been able to explore a lot of topics that do not end with GNNs. For me, the interest on the project stayed fresh throughout the development because I had to tackle diverse topics for every section of the tool. I learned about graph representation methods, graph characterization, how Python projects are developed, how the PyG framework worked, etc. I had to even use web development tools like html, css and javascript in order to build the user interface, which makes use of the chrome web browser to render.

I have to thank our director Sergi for all the guidance and advice he has given us throughout the duration of the project. His level of involvement has helped us in a great amount. I also have to thank Cristina and Carlos, who are responsible for the other 2 parts of the bigger project. Since iGNNspector touches in some degree or another a lot of GNN related topics, I had to ask for advice on their domains, and they have never hesitated to share their respective knowledge when needed. Last but not least, I have to thank Albert for setting up and giving us access to a powerful server. Without it, I would have not been able to experiment with big graph datasets. Personally, I have never used such a capable machine and it has been a pleasure.

9. Budget and sustainability

9.1. Budget

In this section I will describe the economic cost associated with the resources that will be used to complete the project, among them, human resources or hardware resources, for example. Later, a cost table can be found, which shows the cost that each task will have, according to its resources and the total cost of the project. Furthermore, I will explain how I plan to set an economic emergency margin if some deviation were to occur. All costs are expressed in euros (€).

9.1.1. Cost Identification and Estimation

9.1.1.1. Cost per task

First of all, behind every project there is a team of people who will write the planning and push it forward to complete it. In the case of this thesis, this team is comprised by:

- **The thesis researcher, developer, tester and writer.** I will be responsible for carrying out the majority of the tasks that comprise this project. Since the goal consists of developing a software tool to work with GNNs, the work that has to be done is fairly varied.
- **The thesis director,** who will offer guidance, advice, support and will supervise the work done.

9.1.1.2. Generic costs

The field of Machine Learning is well known for the high computational cost that its algorithms require, and GNNs are no exception. In order to be able to run the code from the tool, some hardware is absolutely necessary.

- A **server** with a rather capable GPU and RAM is needed. The BNN-UPC research group has provided us with a server with a powerful GPU that can be accessed via an SSH connection and a VPN.
- A **laptop** will be required to access the server at any time. Although it will not run any algorithm, it will be used to perform all the research and write all the code for the software tool.

The amortization formula used to determine the costs of hardware resources is the following:

$$AC = Price \times \frac{1}{U_T} \times U \qquad (7)$$

Where:

- AC is the amortized cost for the project in €,
- Price is the initial price of the good in €,
- U_T is the total amount of hours the good will be used and
- U is the amount of hours it will be used for the project.

All the hardware components that constitute the server sum up a price of approximately 4,500 €. It will be used by the BNN-UPC research group during a period of approximately 4 years. During those years it will be able to run useful work 24 hours a day every day. The server will be used during the research of the most fitting algorithms to use, implementation and testing. Estimate that it will be used approximately 300 hours. With these numbers, the amortized cost is:

$$AC_{server} = 4,500 \times \frac{1}{4\times24\times365} \times 300 = 38.52\,€ \tag{10}$$

The laptop initial price was 1,160 €. It is currently being used also for personal and academic purposes, so the amount of hours spent on average per day is 4 hours. I intend to use it for at least 4 years, so the total amount of hours which the laptop's will be used (U_T). It is planned to be used during all phases of the project, which sum up to 595 hours. The resulting amortized cost is:

$$AC_{laptop} = 1,160 \times \frac{1}{4\times4\times365} \times 595 = 118\,€ \tag{11}$$

In order to use the hardware mentioned before, an energy source is needed. During the period where the project will be conducted, it is expected that the electricity rate in Spain will sit around 0.12 €/kWh. When running under a high computational load, the server uses 800W of power. However, it will not be at full throttle all the time it is used. I expect it to run at this state only 30% of the time at most, which is 90 hours. The rest of the time I estimate it will run at 300W. Obviously, the laptop is much less powerful, sitting at 250W. The cost of the total energy consumption of the hardware can be calculated as such:

$$EC = price \times P \times T \tag{12}$$

Where:

- EC is the total cost of the energy used in €,
- *price* is the price in €/kWh of electricity
- P is the power from the hardware in kW and
- T is the time it spends consuming that much power in hours.

$$EC_{server} = 0.12 \times (0.8 \times 0.3 + 0.3 \times 0.7) \times 300 = 16.2\,€$$

$$EC_{laptop} = 0.12 \times 0.25 \times 595 = 17.85\,€$$

$$EC_{server} + EC_{laptop} = 34.05 €$$

9.1.2. Cost Table

Macro-Task	**Hours**	**cost per hour (€)**	**Macro-Task cost (€)**
Project management	90	15	1350
Study of the GNN field	50		750
Study of the main programming frameworks	70		1050
Research and determine algorithms	85		1275
Implementation	100		1500
Unit testing	35		525
Tool creation	30		450
Integration testing	20		300
Documentation	120		1800
Total cost per task (€)			**9000**

Hardware	**Hours used**	**Amortized cost per hour (€)**	**Total amortized cost (€)**
Server	300	0.863	38.52
Laptop	595	0.198	118.18
Total cost per task (€)			**156.7**

Hardware	**Hours used**	**Electricity cost**	**Total electricity cost (€)**
Server	300	0.12	16.2
Laptop	595		17.85
Total cost per task (€)			**34.5**

Total project cost (€)		**9191.2**

9.1.3. Management Control

It is very recomended to think about the planning of a project as if contingencies and complications were totally expected to happen, as if they were programmed, only that without a specific date to address them. To further understand what these deviations from the original plan look like once the project is done, I will list next a set of metrics that will help to visualize where contingencies will have happened and the magnitude of them.

- **Human resources deviation.** There are a lot of complex algorithms related to GNNs that need to be implemented. Also, the tool to develop will be made up of multiple software packages and pieces. It is possible that there is more complexity to the project than what we originally considered. Which means that more hours will be needed to finish some aspects of the implementation. It is also possible that algorithms are more computationally complex than what we envisioned. These facts can contribute to slowing down development.

$$HR_d = \sum_{p \in P} (eu_p - ru_p) \times C_p \tag{13}$$

Where:
- HR_d is the human resource cost deviation,
- P is the set of people working in the project,
- eu_p is the estimated amount of hours that a person p will work in the project,
- ru_p is the real amount of hours that a person p will work in the project,
- C_p is the cost per hour of a given person p.

- **Amortization deviation.** As mentioned before, additional working hours mean additional hours that hardware equipment is used and amortized.

$$Am_d = \sum_{i \in I} (eu_i - ru_i) \times C_i \tag{14}$$

Where:
- Am_d is the amortization cost deviation,
- I is the set of hardware used in the project,
- eu_i is the estimated amount of hours that a hardware i will work in the project,
- ru_i is the real amount of hours that a hardware i will work in the project,
- C_i is the cost per hour of a hardware i that will be used in the project.

- **Total cost deviation.**

$$TC_d = ec - rc \tag{15}$$

Where:
- TC_d is the total cost deviation,

- ec is the estimated general cost of the project and
- rc is the real cost of the project.

9.2. Sustainability

9.2.1. Economic Dimension

Regarding PPP: Reflection on the cost you have estimated for the completion of the project:

Because of the current situation, the costs related to transport have been entirely cut since we will work on the project from our homes. Also, no hardware equipment has been bought specifically for this project. The server has been and will be used for a lot more projects from the BNN-UPC research group. The laptop is also used for personal and academic purposes.

Regarding Useful Life: How are currently solved economic issues (costs...) related to the problem that you want to address (state of the art)? How will your solution improve economic issues (costs ...) with respect to other existing solutions?

Nowadays, the techniques used in the field of GNNs require a substantial amount of computing power. That fact forces researchers and companies to spend a substantial amount of money on capable computers. Moreover, the performance and efficiency of a GNNs is tightly bound to the properties of the graph that wants to be analyzed. Because of this, researchers have to spend a considerable amount of time testing which algorithms work best with which types of graphs. The thesis aims to develop a tool that can tell to its users what algorithms work best and give the most efficient models possible, in order to save costs related to equipment and time.

9.2.2. Environmental Dimension

Regarding PPP: Have you estimated the environmental impact of the project?

Given the current situation related to the pandemic, all the work will in principle be carried out remotely. Since everyone can work from their home, we will cut a lot of our carbon footprint. With respect to the source of the electricity used to run the hardware, unfortunately I do not have any source of 100% renewable energy, like solar panels. The carbon footprint related to the hardware is tight to the proportion of power generation methods that are currently in use at the areas where we live and where the server runs at.

Regarding PPP: Did you plan to minimize its impact, for example, by reusing resources?

I am currently using for the project a laptop that I also use for personal and academic purposes. Since the main computations are carried out at the server, if something happened to the laptop I would reuse another laptop to avoid having to buy a new one. With regards to the server, it will be used long after the end of the project to amortize the investment and minimize the carbon footprint.

Regarding Useful Life: How are currently solved economic issues (costs...) related to the problem that you want to address (state of the art)? How will your solution improve economic issues (costs ...) with respect to other existing solutions?

As mentioned at the Economic Dimension section, GNNs are expensive to compute. Computationally expensive algorithms require a lot of energy to run at servers. Depending on the location of the servers this consumption can translate to a lot of carbon emissions. Also, the majority of the emissions caused by a hardware component are related to production. Computers which are already some years old might be replaced by newer ones in a shorter time frame, contributing to more carbon emissions. By providing researchers and professionals of the field with an optimized model for their specific needs, there is less need for new hardware and time to test and run algorithms, which leads to less emissions.

9.2.3. Social Dimension

Regarding PPP: What do you think you will achieve -in terms of personal growth- from doing this project?

Firstly, I will gain experience working in a team project and will receive useful advice from my director, who already has experience working in a research team. Also, myself and two more students will help each other as we research different aspects from the GNN field. Moreover, I will get to know a lot from an emerging field in Machine Learning, which I think will help me if I decide to follow this career path. Finally, I will learn a ton indirectly from this thesis, as I will need to use frameworks and software that I never used before in order to construct the tool.

Regarding Useful Life: How is currently solved the problem that you want to address (state of the art)? How will your solution improve the quality of life (social dimension) with respect to other existing solutions?

The field of this thesis, GNNs, is used in sectors where Machine Learning algorithms already operate in some form or another. Nonetheless, GNNs show a lot of potential improvement regarding the analysis of real world graph based data. This kind of data is very diverse. For example, we find data related to medicine, biology, physics, climate, transportation, economics, social interaction, etc. This means that there is also potential for improvement in all of these other fields that can lead to improvement in health, transportation, environmental solutions, economics, etc.

Regarding Useful Life: Is there a real need for the project?

This work can help in some proportion to accelerate the pace at which GNNs are developed, tested and deployed, this can enable a wider adoption of GNNs by different industries, which in turn can accelerate improvement in all the fields mentioned above.

9.2.4. Self assessment

This self assessment from the survey made me think that, while I have a general understanding about sustainability, and social commitment, I lacked the knowledge from a more technical approach to these issues. I am inclined to steer a project towards sustainable solutions and good social impact, but as I reflected on the issues I became more aware that I actually did not know the tools and metrics to actually tackle these problems, or it had been a long time since I had to do some analysis.

I know that other students will think like that. This section helped me refresh some concepts as well as teach some new ones related to the planning and analysis of the social, environmental and sustainable part of a project.

10. References

[1] S. Abadal, A. Jain, et al, “Computing Groah Neural Networks: A Survey from Algorithms to Accelerators,” (2020). URL: https://arxiv.org/abs/2010.00130

[2] A gentle introduction to Graph Neural Networks (Basics, DeepWalk, and GraphSage) URL: https://towardsdatascience.com/a-gentle-introduction-to-graph-neural-network-basics-deepwalk-and-graphsage-db5d540d50b3

[3] Battaglia, Peter W., et al. "Relational inductive biases, deep learning, and graph networks." arXiv preprint arXiv:1806.01261 (2018). URL: https://arxiv.org/abs/1806.01261

[4] Tutorial 7: Graph Neural Networks. URL: https://uvadlc-notebooks.readthedocs.io/en/latest/tutorial_notebooks/tutorial7/GNN_overview.html

[5] Thomas N. Kipf, Max Welling, “Semi-Supervised Classification with Graph Convolutional Networks,” arXiv preprint arXiv:1609.02907, (2017) URL: https://arxiv.org/abs/1609.02907

[6] Pytorch Geometric Documentation, TORCH_GEOMETRIC.NN, Convolutional Layers. URL: https://pytorch-geometric.readthedocs.io/en/latest/modules/nn.html#convolutional-layers

[7] Petar Veličković, Guillem Cucurull, et. al, “Graph Attention Networks,” arXiv preprint arXiv:1710.10903, (2018) URL: https://arxiv.org/abs/1710.10903

[8] R. Garg “A Taxonomy for Classification and Comparison of Dataflows for GNN Accelerators” (2021) URL: [2103.07977] A Taxonomy for Classification and Comparison of Dataflows for GNN Accelerators (arxiv.org)

[9] Pytorch Geometric Documentation. URL: Introduction by Example — pytorch_geometric 1.6.3 documentation (pytorch-geometric.readthedocs.io)

[10] NetworkX Documentation. URL: https://networkx.org/documentation/stable/

[11] The Kanban method in IT development projects. URL: https://www.bocasay.com/kanban-method-it-development-projects/

[12] M. Gori, et. al, “A New Model for Learning in Graph Domains”, (2005) URL: https://www.researchgate.net/publication/4202380_A_new_model_for_earning_in_raph_domains

[13] F. Scarselli, et. al, “The graph neural network model”, (2009) URL: https://persagen.com/files/misc/scarselli2009graph.pdf

[14] B. Knyazev, “Tutorial on Graph Neural Networks for Computer Vision and Beyond” URL: Tutorial on Graph Neural Networks for Computer Vision and Beyond | by Boris Knyazev | Medium

[15] C. Tian, et al. “PCGCN: Partition-Centric Processing for Accelerating Graph Convolutional Network”, (2020) URL: PCGCN: Partition-Centric Processing for Accelerating Graph Convolutional Network | IEEE Conference Publication | IEEE Xplore

[16] Z. Jia, et al. “Redundancy-Free Computation for Graph Neural Networks” (2020)URL:Redundancy-Free Computation for Graph Neural Networks (stanford.edu)

[17] Z. Jia, et al. “PairNorm: Tackling Oversmoothing in GNNs” (2020) URL:1909.12223.pdf (arxiv.org)

[18] M. Chen, et al. “Simple and Deep Graph Convolutional Networks” (2019) URL:Simple and Deep Graph Convolutional Networks (arxiv.org)

[19] G. Li, et al. “DeeperGCN: All You Need to Train Deeper GCNs” (2020) URL:[2006.07739] DeeperGCN: All You Need to Train Deeper GCNs (arxiv.org)

[20] K. Zhou, et al. “Towards Deeper Graph Neural Networks with Differentiable Group Normalization” (2020) URL:[2006.06972] Towards Deeper Graph Neural Networks with Differentiable Group Normalization (arxiv.org)

[21] GML Newsletter: Homophily, Heterophily, and Oversmoothing for GNNs URL:GML Newsletter: Homophily, Heterophily, and Oversmoothing for GNNs - Graph Machine Learning (substack.com)

[22] H. Pei, et al. “Geom-GCN: Geometric Graph Convolutional Networks” (2020) URL:2002.05287.pdf (arxiv.org)

[23] J. Zhu, et al. “Beyond Homophily in Graph Neural Networks: Current Limitations and Effective Designs” (2020) URL:[2006.11468v2] Beyond Homophily in Graph Neural Networks: Current Limitations and Effective Designs (arxiv.org)

[24] M. Fey, et al. “Fast Graph Representation Learning with PyTorch Geometric” (2019) URL:[1903.02428v1] Fast Graph Representation Learning with PyTorch Geometric (arxiv.org)

[25] iGNNition URL:Welcome to IGNNITION — ignnition main documentation

[26] W. Hu, et al. “Open Graph Benchmark: Datasets for Machine Learning on Graphs” (2021) URL:2005.00687.pdf (arxiv.org)