Research Project

2n Batxillerat

On non-static graphs

- Graph-algorithm interaction with machine learning -

Presented by Eloi Pagès Vives— June 28, 2022

"This question is so banal, but seemed to me worthy of attention in that neither geometry, nor algebra, not even the art of counting was sufficient to solve it."

Leonard Euler

Abstract

Graph Theory, the branch of mathematics and computer science that studies the properties of graphs -an abstract representation of reality- and its algorithms. From a mathematical perspective, in the first part of the article we rediscover the fundamental properties of graphs and some of the essential traversing algorithms that are omnipresent in our daily lives. Spanning from basic theorems to trees and specific properties; this research gives a general idea of the role of mathematics in graph theory and an overview of the the most famous computational problems. At first glance, it seems that this field of study, which brings mathematics and computer-science together, is nothing but limitless. However, as strange as it may seem, this field faces a major limitation: once a graph has been defined, it remains constant. To face this problem, some well-known machine learning algorithms are combined with knowledge of Data Science to develop a new interactive way of interpreting a graph. Instead of defining an edge as a constant element of a set, it is defined through a series of functions which will be called *generator* and *error* functions. After the formal definitions required and some analysis on this new family of graphs, graphs are applied to the real world. The first application, concerning traditional graphs, is related to the sustainable development objectives and aims to find an optimal connection between cities in an underdeveloped region so they can further develop their economies. The practical application of non-static graphs lays in finding the shortest path between two locations in a city whilst taking traffic into account. Using this newly described version of a graph, a modification of Dijkstra's algorithm is run on an interacting non-static graph which yields the time taken to travel from the desired cities.

Resum

Teoria de graphs, la branca de les matemàtiques i la computació que estudia les propietats dels graphs i els seus algorismes, una manera fascinant de representar la realitat. Vist des d'un punt de vista matemàtic, a la primera part de l'article es comprenen les propietats fonamentals dels graphs i els algorismes que estan omnipresents a les nostres vides. Anant des de teoremes bàsics fins a arbres i propietats específiques, es dona una idea general de la teoria de graphs i una anàlisi superficial dels problemes computacionals més coneguts. Tot i que sembla estrany aquest àmbit té una limitació considerable: un cop es defineix un graph, aquest es queda constant durant tot el procés d'anàlisi. Per enfrontar aquest problema, utilitzo algorismes de machine learning per desenvolupar una manera interactiva d'interpretar els graphs. En lloc de definir una aresta com a constant, és interpretada amb una sèrie de funcions que es diran funcions d'*error* i *generadores*. Després de les definicions formals que requereix aquesta nova teoria, els graphs no estàtics són aplicats a la vida real. L'aplicació dels grafs tradicionals és trobar la conexió òptima entre un conjunt de ciutats d'un país subdesenvolupat, afavorint així que desenvolupin les seves economies. L'aplicació de grafs no estàtics és torbar el camí més curt d'una ciutat a un altre tenint en compte el tràfic. Emprant aquesta nova versió del graphs, una modificació d'un Dijkstra és utlitzada en els graphs no statics que ens acaba donant el temps òptim per anar de citat a ciutat.

Gratitude

I want to express my deepest gratitude to my brother, for revising the rigour of the mathematics and most importantly for the orientation given on how to develop section 7.

Table of contents

1 Introduction **8**

1.1 Motivations 8

1.2 Formal Objectives 9

1.3 An overview of the objectives of each section 10

1.4 Personal objectives 11

1.4.1 Limitation of the research 12

1.5 NOTE 12

2 Graph Theory **14**

2.1 Graph Terminology 14

2.2 Graph representation 16

2.2.1 Visual representation of graphs 16

2.2.2 Matrices 17

2.2.3 Adjacency list 18

2.3 Traversing a graph with Depth First Search 18

2.3.1 Procedure 19

2.3.2 Complexity analysis 21

2.4 Types of Graphs and its unique properties 22

2.4.1 Connected/disconnected Graphs 22

2.4.2 Complete graphs 23

2.4.3 Cyclic graphs 24

2.4.4 Trees 24

2.5 The shortest path problem 26

2.5.1 Breadth's First Search, finding the shortest path in an unweighed graph . 26

2.5.2 Dijkstra's algorithm, finding the shortest path 29

2.5.3 Optimizing Dijkstra's algorithm for sparse graphs 32

2.6 Graphs to model real life situations 34

2.6.1 Interpretation of the graph 34

2.6.2 Construction of a graph 35

3 Non-static graphs, definitions and theoretical analysis **37**

3.1 Graph limitations . . . 37

3.2 A limited problem example, Shortest path . . . 37

3.3 The generalized concept . . . 38

3.4 Deleting edges, the error function . . . 38

3.5 Representation of a non-static graph . . . 40

3.5.1 Adjacency matrix and concrete configurations . . . 40

3.5.2 Three dimensional representation of a non-static graph . . . 41

3.5.3 Considering arbitrary generator functions as an example . . . 42

3.6 Non-static trees, an error function example . . . 43

3.6.1 Disjoint Set Union, DSU . . . 44

3.6.2 Constructing the MST for each configuration . . . 46

3.6.3 Proof of the validity of Kruskal's greedy algorithm . . . 47

4 Non-static graphs as an abstract model **48**

4.1 Required dataset . . . 48

4.2 Simple linear tendency . . . 50

4.2.1 Minimization of the error squared . . . 51

4.2.2 Implementation of the algorithm . . . 54

4.2.3 Resulting graph . . . 55

4.2.4 Pearson correlation to check the correctness of the generator function . . . 57

4.3 Multi-variable linear tendency . . . 58

4.3.1 An expansion on error squared for m dimensions . . . 59

4.3.2 Gauss-Jordan elimination . . . 62

4.3.3 The complete algorithm . . . 65

4.3.4 Resulting graph . . . 66

4.3.5 Multiple correlation coefficient . . . 67

4.4 A new approach to the Multiple Regression . . . 68

4.4.1 Prove of the validity of this approach . . . 69

4.4.2 Polynomially transforming a vector . . . 70

4.4.3 Computation of the polynomial degree . . . 71

4.4.4 The predicted edge function 72

5 Modular implementation of a non-static graph 73

5.1 Edge Class 73

5.2 Graph Class 75

6 Connecting underdeveloped civilizations in an efficient way 79

6.0.1 Overview of the problem 79

6.0.2 Building the graph 80

6.0.3 Final results 83

6.0.4 How to improve on this method 83

7 Finding the shortest path with a non-static graph 85

7.1 Interpreting the situation 85

7.2 The weight of each edge and generator function 87

7.2.1 The weight for small voyages 88

7.2.2 Generator function for long distances 89

7.3 Error function 92

7.4 The algorithm 94

7.4.1 Standing still in a vertex 94

7.4.2 Process 95

7.4.3 Evaluation of the interaction between the graph and the algorithm 95

7.4.4 Complexity 96

7.5 Final results 97

8 Conclusions and ending 98

8.1 Traditional Graphs 98

8.2 Non-static graphs 99

8.3 Connecting Cities 99

8.4 Shortest path with traffic 100

8.5 How to further on the investigation 100

8.6 Ending 101

References **102**

9 Annex A, Further information **105**

9.1 Big O notation 105

9.2 Binary Operators 106

9.3 Bipartite graphs, additional information 106

9.4 An interesting arithmetic proof using complete graphs 109

10 Annex B, used applications **109**

10.1 LaTeX 110

10.2 Wolfram Mathematica 110

10.3 Tikzpicture 110

10.4 Desmos 111

10.5 C++ 111

1 Introduction

1.1 Motivations

I have been passionate for computer science and maths since I was a kid. This passion lead to an ambition to program at the higher nation level, the Spanish Olympiad on Informatics (OIE). Because of this I got to train with some of the best teachers in the country. It was there that I took a glimpse of graph theory, it was love at first sight. Not only was I taken aback by the endless possibilities that such a simple interpretation of reality offered but also stunned by the limitless computational opportunities that such theory offered.

Moreover, I have always liked data-science and artificial intelligence. One year before, I had been reading a marvellous book on artificial intelligence called *Artificial intelligence, A modern approach*[RN02] by Stuart Russell and Peter Norvig . This book made me realise that machine learning is the future of technology and encouraged me to apply to an on-line course about programming machine learning. This got me a grasp of the field, and because the course only dealt with the basic principles and a trivial implementation through libraries, I got encouraged to pursue my learning.

It was around this time where I had to plan my research project for the upcoming year. My first idea was to produce a philosophical essay about the relationship between the learning process of a human and the learning process of a machine. However, this project got quickly discarded when, in math class, we started getting a taste of function analysis. After some days on the topic, whilst I was practising graph algorithms I asked myself the mundane question of "What would happen if the weights of the edges were functions and not constants?". This question came from the fact that i thought that the existing graph theory did not allow to find the shortest path from one point to another one whilst taking traffic into account. When traffic is considered, the edge weights become functions and every single aspect of graph theory fails.

This simple question further developed into an obsession and I decided to change my entire project to research it. It was in the middle of my research that I had the idea to include machine

learning in the project. This last addition united my 3 favourite fields.
As the language regards, just like I will say in Objectives, I have written this paper in English not only to practice my language but also to train my formal writing of science, because it will be in English that I will write all the professional publications that I will do in my career as a scientist.

1.2 Formal Objectives

The general objective of the project is to answer the question raised in motivations: "What would happen if the edges of a graph were functions?" and try to define a new type of graph that allow to find the shortest path whilst taking into account traffic. This will be done through the definition of non-static graphs.

The first objective of this project is to rediscover the important graph properties that were first proven nearly two centuries ago. The knowledge of these theorems will allow me to understand all the algorithms that are often used to process graphs and prove why they work.

With the applicable essence of graph theory in mind, this research aims to elaborate an expansion on graph theory to make them more applicable to the real world and allow and therefore enable them to solve the route planning problem. To do so, it aims to redefine how graphs are interpreted creating a new type of graph which will be baptised as non-static graph.

Keeping this in mind, this article aims to formally establish the bases of non-static graphs and raise what could be an innovative new way to use this incredible data structure. To further on that it also aims to define a general process on how these newborn graphs could be used to model real life situations.

At the end, this project aims to raise two viable projects that are solved using traditional graphs and non-static graphs respectively. If possible, the traditional graph project should be related the the sustainable development goals established in 2015 because I think that it is really important to be conscious about them and I would like to put my grain of sand. On the other hand, the

non-static graphs application will try to solve the problem that encouraged me to create this theory, finding the shortest path whilst taking traffic into account.
Computationally speaking this research aims to describe already exiting algorithms and to implement them in C++by myself. It also aims to design and prove new algorithms to solve the problems that are encountered through development of non-static graphs.

1.3 An overview of the objectives of each section

To finish with the objectives of this research, I want to point out what aims to do each important section of this monograph.

Graph Theory: This first part of the monographs aims to define the most important concepts of graph theory and prove the most famous properties from a mathematical point of view. Moreover it also aims to describe the better know traversing algorithms and solve the shortest path problem in the traditional way.

Non-static graphs, definitions and theoretical analysis: This introduction to a new concept that will be called non-static graphs aims to define what they are and to evaluate the key points of them: visual representation, generator functions and error functions.

Non-static graphs as an abstract model: After everything is defined, this section aims to describe a general idea on how non-static graphs could be used to model real world situations.

Modular implementation of a non-static graph: This last theoretical part discusses the modular implementation in C++of a non-static graph. All the concepts exposed in the previous sections are put together in pseudo-code to give a general idea on how to implement this type of graph using Classes.

Connecting underdeveloped cities: This first practical application is an example of a problem that can be solved using traditional graphs. Using a simple MST algorithm, a program finds and optimal way to connect different cities in an underdeveloped region so they can develop their economies and break the cycle of poverty.

Optimizing route planning for long trips: This last section is the part that puts together the three sections of non-static graphs and the shortest path problem to a practical application. We discuss the problem of finding the shortest path of long trips whilst taking into account traffic. Using non-static graphs, statistics and Dijkstra's algorithm, we approach the way in which Google Maps computes the optimal route to follow.

Annex A Further information: This first Annex is a compilation of useful information for the reader related to the topic that was not essential enough to put in the main body of the monograph but can result to be useful for the unfamiliar with some topics such as Binary operators and big O notation. It also contains some additional information about traditional graphs, Bipartite graphs.

1.4 Personal objectives

As my personal objectives, I hope that working on this project will make me a better algorithmic programmer and give me further knowledge on computer science. Just as I said in motivations, my main personal objective is to learn more in the rising field of machine learning and to try, even if it means to fail, to apply machine learning to graphs.

One of the reasons why I am doing this project in English is because I am studying for the proficiency exam and I hope that writing this many pages in the language has helped me to get better on the formal writing aspect of English. Moreover, writing a formal scientific journal in English will help me prepare for all the future academic papers that I will have to write, because they will for sure be in this language.

This project has been written completely with the LaTeX program. A programming language that compiles to pdf and allows to create beautiful math papers. I also aim to end this project with a complete dominance of the LaTeX language and to be able to use it with my everyday school assignments and the future academic papers that I will hopefully write in university and beyond. Regarding the same topic I also aim to get a hand on the Wolfram Mathematica [Wol91] language which is an essential tool for every aspiring mathematician, physicist, computer

scientist... It allows to create unbelievable graphics, animations and do lots of many other things. In this project it was only used for 3D graphics and getting coordinates of an image.[1]

1.4.1 Limitation of the research

This research has two major limitations. At first, the machine learning algorithms that I would like to use are too hard to implement, so I have settled into simpler ones. Secondly, since nowadays the data is so private, it is really hard to get your hands on lots of it. This has made me use computer generated datasets for the purpose of testing.

1.5 NOTE

All the images that can be seen in this monograph except from figure 20 have been computationally generated by the author of the monograph and belong to it. To see the programs used, see Annex B.

[1] I want to specially mention this two programs because without them this project would not have been possible

"Written so that readers unfamiliar with physics may not feel like the wanderer who was unable to see the forest for trees"

A.Einstein

This page has been intentionally left blank

2 Graph Theory

Graph Theory is a branch of mathematics and computer science defined as the study of properties of graphs and its algorithms.

A graph is a mathematical representation of the relations between some elements (nodes). Formally speaking, a graph $G = (V, E)$ of size (n, m), where $n = |V|$ and $m = |E|$, is defined by a set of vertices or nodes $V = \{v_1, v_2, v_3, \ldots, v_n\}$ related in some way by a set of edges $E = \{e_1, e_2, e_3, \ldots, e_m\}$. Every edge is distinguished by the two vertices it connects and an optional parameter[2] w which represents the weight[3] of the edge, $e_{ij} = \{v_i, v_j, w_{ij}\}$[4] $v_i, v_j \in V$. The edges can be noted in two different ways, as e_i, we take the i^{th} edge of the set mentioned above, and as e_{ij}, meaning that we take the edge that connects $v_i \to v_j$.

2.1 Graph Terminology

Having stated the definition of a graph, it is necessary to present some terminology that will be used throughout this article. First of all, it is important to define three functions which allow us to access the three properties of an edge, $s(e_{ij})$, $t(e_{ij})$ and $w(e_{ij})$. Given an edge $e_k \in E$, we define the source of the edge as $v_i = s(e_{ij})$ the target as $v_j = t(e_{ij})$ and the weight as $w_k = w(e_k)$. It is important to note that in a weighted graph we treat $w(e_{ij}) = \infty \ \forall e_{ij} \notin E$ and that if $w(e_{ij}) = k \ \forall e_{ij} \in E$ we may consider the graph to be unweighed.

The edges of a graph have one last characteristic, the direction. An edge e_{ij} is said to be directed if $w(e_{ij}) \neq w(e_{ji})$ otherwise, the edge is undirected. Formally, we say that a graph G is an undirected graph if $w(e_{ij}) = w(e_{ij}) \ \forall e_{ij} \in G$ and a directed graph or digraph otherwise. Visually, the directed edges of a graph are represented with an arrow, from $i \to j$ and the undirected ones with a normal line. It is easy to see that a directed graph is more general that an undirected one, consequently all algorithms that work for directed graphs will also work on undirected ones but the opposite is not always true. (Note that if an edge is directed $v_i \to v_j$, while traversing

[2] w is only mandatory in a weighted graph. Otherwise, it is defined as a constant in all E

[3] The weight of an edge is the "effort" it takes to go through

[4] The edge e_{ij} would connect the vertex v_i with v_j with weight of w_{ij}

the graph, this edge can not be taken from $v_j \to v_i$).

Having stated all of the previous, I will present some of the most important concepts/definitions that will be used in the following sections:

1. We say that a vertex v_i is *adjacent* to v_j if there $\exists\ e_k \in E$ such that $s(e_k) = v_i \wedge t(e_k) = v_j$. In undirected graphs, if v_i is adjacent to v_j, v_j will be adjacent to v_i. For example, in the undirected graph G_A (see figure 1) we can say that 2 is adjacent to 4 and 4 is adjacent to 2 whereas in the directed one G_B, this property does not hold for A and B.

2. In $G = (V, E)$ we express the number of vertices as $|V|$ and the number of edges as $|E|$, usually using n and m respectively.

3. An edge e_{ij} is *incident* to v_i and v_j in undirected graphs and to v_j in directed ones.

4. A vertex can present two types of *degree*, the *inner degree* $\delta^+(v_i)$, and the *outer degree* $\delta^-(v_i)$. The inner degree of v_i is the number of edges that connect a vertex to v_i and the outer degree is the number of edges that connect v_i to any vertex.

5. Having defined degrees, in a directed graph, we call *source* a vertex which has an inner degree of 0 and *sink* a vertex which has an outer degree of 0. And in any graph, an *isolated* vertex v_i is a node where $\delta^+(v_i) = 0$ and $\delta^-(v_i) = 0$. However, when traversing a graph, we also call source the vertex from which the algorithm has been initiated.

6. A *self-loop* is an edge that connects a vertex v_i with itself. (see 1, vertex B) We also say that there is a *multiple edge* between v_i and v_j if there exists more than one edge that connects the two vertices.

7. If there are neither self-loops nor multi-edges in a graph, it is called *simple graph*, otherwise it is a *multi-graph*.

8. A path $\rho(v_i, v_j) = \{v_i, e_{ia}, v_a, e_{ab} \dots e_{kj}, v_j\}$ is a sequence of alternating vertices and edges where there are no repeated vertices nor edges.

9. We call *bridge* an edge e_k which its deletion would add one component to the graph.

10. A graph $G' = (V', E')$ is said to be a *subgraph* of G if for all v_i in V', v_i is also in V and for all $e_i \in E'$, e_i is also in E. in other words, $V' \subset V$ and $E' \subset E$.

11. *Graph traversal* is the process of searching through a graph to explore the properties by visiting and updating each vertex in the graph. Depending of how we traverse the graph, we will be able to store and find different properties of the explored graph.

12. An *anonymous graph* is a graph where its vertices have not been given a name, so it is impossible to distinguish them from the others. These graphs are used for illustration purposes because they are clearer to the naked eye.

2.2 Graph representation

Graphs can be represented by drawings (see 2.2.1) or by data structures (see 2.2.3 or 2.2.2). For all notations, every node in a defined graph will be noted with a number $x \in \mathbb{N}$ or a capital letter $\alpha \neq E$[5]

2.2.1 Visual representation of graphs

Drawing a graph is normally done by depicting every node v_i with a circle with the name of the note inside. After this, according to the configuration of the graphs, the edges e_{ij} are drawn using a simple line v_i — v_j in undirected graphs and an arrow $v_i \rightarrow v_j$ in directed graphs. To complete the graph, in case of a weighted graph, every weight is written above the respective edge.

Anonymous graphs are drawn with smaller nodes filled with black (see figure 9.) These type of graphs are come in handy when their only intention is to illustrate a concept and the specific vertices do not need to be referenced in the text.

During the following sections we will be using G_A and G_B to show the different notations so keep in mind these two images.

[5] E will be skipped to avoid confusions with the set E

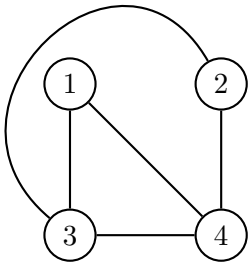

Figure 1: G_A

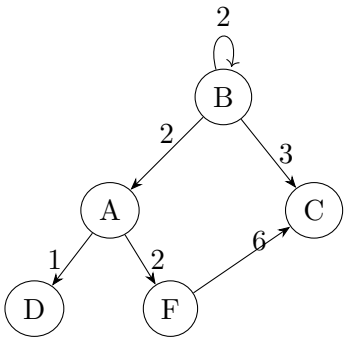

Figure 2: G_B

2.2.2 Matrices

Matrices are the first data structure that is going to be considered to store graphs. As an advantage, matrices come with the possibility of computing whether an edge $e_{ij} = \{v_i, v_j\}$ is in E in $O(1)$ complexity. However, this comes with a major disadvantage, matrices take $O(n^2)$ space complexity[6] and $O(n^2)$ to iterate through, which is not optimal and will be unable store nor process graphs where $|V|$ is bigger than $\approx 10^4$.

The most popular matrix representation and the only one that will be used in this article is the adjacency matrix. The adjacency matrix A of a Graph $G = (V, E)$ is a $|V| \times |V|$ matrix where:

$$A_{ij} = \begin{cases} w(e_{ij}) & \text{if } e_{ij} \in E \\ 0 \text{ or } \infty & \text{if } e_{ij} \notin E \end{cases} \tag{1}$$

This matrix has has a space/time complexity of exactly $O(|V|^2)$, as to represent the matrix we need to store $|V|$ columns and rows. Furthermore, the adjacency matrix has the unique ability to preserve both the weight and the direction of the edges. For undirected graphs, this matrix holds the property that it is symmetric on the $(0, 0)$, (n, n) diagonal, so, $A_{ij} = A_{ji}$ as can be seen in Figure 3.

[6]Too see that big $O()$ notation is, see Annex A, Further information **9.1 Big O Notation.**

$$\begin{pmatrix} 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{pmatrix}$$

Figure 3: Adjacency matrix of G_A

$$\begin{pmatrix} \infty & \infty & \infty & 1 & 2 \\ 2 & 2 & 3 & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 6 & \infty & \infty \end{pmatrix}$$

Figure 4: Adjacency matrix of G_B

2.2.3 Adjacency list

Adjacency lists are the most used data structure to store graphs because, contrarily to matrices, they take a linear $O(|V|+|E|)$ space complexity, being able to store graphs up to $|V|+|E| \approx 10^8$. An adjacency list ADJ is a set of arrays a_i such that $ADJ = \{a_1, a_2, \ldots, a_{n-1}, a_n\}$ where $n = |V|$ and each array a_i consists of all the vertices that are adjacent to v_i. Contrarily to Matrices, computing whether a vertex v_j is adjacent to v_i runs in $O(|V|)$ because ADJ_{ij} is not always v_j, thus, the entire array a_i has to be iterated. However, iterating the entire graph only takes a linear time of $O(|V| + |E|)$, which is considerably much faster than the quadratic time matrices take. It is important to note that if all possible edges e_{ij} belong to the graph (the graph is complete) the adjacency list becomes the adjacency matrix. We can therefore conclude that as the graph grows in size, the matrix option becomes more viable.

v_1 [1] → [4][3]
v_2 [2] → [3][4]
v_3 [3] → [1][2][4]
v_4 [4] → [1][2][3]

Figure 5: Adjacency list of G_A

v_1 [A] → [D][F]
v_2 [B] → [A][B][C]
v_3 [C] → |
v_4 [D] → |
v_5 [F] → [C]

Figure 6: Adjacency list of G_B

2.3 Traversing a graph with Depth First Search

Depth First search, or DFS [Fie73], first invented by the french mathematician Charles Pierre Tremaux with the intention to use it for solving mazes, is the main algorithm for general graph

traversal because with minor modifications, it allows us to find a great deal of properties in a very efficient matter.

2.3.1 Procedure

The algorithm will visit the vertices in an arbitrary order. It starts from a vertex which we will call source s and goes as deep as possible in the graph, marking all traversed vertices as visited. When it can not go any deeper it backtracks to the last visited vertex and repeats the process until all vertices have been marked.

To make the algorithm work, we first initialize the vector $visited[i]$ which will store whether a vertex v_i is marked (it has already been visited). Since at the start not a single vertex has been visited, $visited[i] = false$ for all i. Throughout the algorithm, as the vertices are visited, this vector will be updated accordingly and will be used to avoid repeating the vertices, which will avoid an infinite loop and yield us a linear complexity. Furthermore, if at the end of the algorithm, $visited[j] = false$, we know that v_j is not reachable from the source.

Secondly we have to initialise a stack called *pending*. A stack is a data structure which works similarly to a vector, and will allow us to store the vertices that are pending the evaluation. The data structure has two main functionalities, *Push*, a vertex, which will put the chosen vertex on top of the stack, and *retrieve* a vertex, which will pick the top element of the stack. At the start of the algorithm, the stack holds one element, the source.

With these data structures initialized, we can proceed to start the algorithm. DFS will work inside a loop that will run until *pending* is empty. In every iteration we *retrieve* v_i from the top of the stack and delete it from it. We mark v_i as visited, $visited[i] = true$ and iterate through all v_j adjacent to v_i. In reach iteration, if $visited[j] = false$, it is pushed *push* to the stack.

A more intuitive approach would be implementing the DFS recursively. To do so, we define the function DFS, which accepts one parameter i, the vertex that the algorithm is evaluating. It first marks v_i as visited ($visited[i] = true$). Then, if goes through all adjacent vertices v_j to v_i

and if $visited[j] = false$, it will call itself recursively with j as the argument.

As can be seen, this algorithm by itself does not have a purpose, but some minor modifications to it, will be used in the following sections to find properties such as cycles, components and paths.

Algorithm 1 basic iterative DFS

Input: adj$\leftarrow \{e_1, e_2, \ldots, e_{m-1}, e_m\}$

```
Declare: visited,path
while stack not empty do
    v_i ← path.retrieve()
    visited[i]← true
    for v_j in adj[v_i] do
        if not visited[v_j] then
            path.push(v_j)
        end if
    end for
end while
```

The recursive approach is a little bit more intuitive because every time $DFS(j)$ is called from $DFS(i)$ it means that we are traversing the edge e_{ij}. However, recursivity takes a lot of memory and is considerably slower than the iterative approach .

Algorithm 2 basic recursive DFS

Input: adj$\leftarrow \{e_1, e_2, \ldots, e_{m-1}, e_m\}$

```
Declare: visited.
functionDFS(v_i, id)
    visited[v_i]← true
    for v_j in adj[v_i] do
        if not visited[v_j] then
            DFS(v_j)
        end if
    end for
end function
```

Output: No output.

2.3.2 Complexity analysis

Having understood how the algorithm works, the complexity has now to be analysed. It is clear that we will only visit each vertex once, so the minimum iterations are $|V|$. However, for every vertex v_i visited, we will traverse all of its concurrent edges e_{ij} because we must check if v_j is visited or not. Since every vertex will be visited only once and all their concurrent edges will be traversed once, we will do at maximum $|V| + |E|$ iterations (in both approaches). This yields us a linear complexity of $O(|V| + |E|)$ which is blazingly fast for sparse graph. However, if the number of edges is very high, the algorithm becomes significantly slower.

2.4 Types of Graphs and its unique properties

2.4.1 Connected/disconnected Graphs

Since I have not defined what a connected [Fie73] graph is yet, this section will be started with it. Two nodes are connected if $\exists\rho(v_i, v_j)$ or $\exists\rho(v_j, v_i)$. Therefore, a graph is called to be a connected graph if $\exists\rho(v_i, v_j)$ or $\rho(v_j, v_i)$ $\forall v_i, v_j \in V$, otherwise it is disconnected. By definition, a disconnected graph can be partitioned into k different subsets $V_1, V_2, \ldots, V_k$ such that every subset $V_a = \{v_1, v_2, \ldots, v_r\}$ complies with the definition mentioned above and $\nexists\rho(v_i, v_j)$ nor $\rho(v_j, v_i)$ $\forall v_i \in V_k,\ v_j \in V \setminus V_k$. These subsets are called components and are the largest connected subgraph of G. In figure 7 the disconnected graph with three components G_C can be seen.

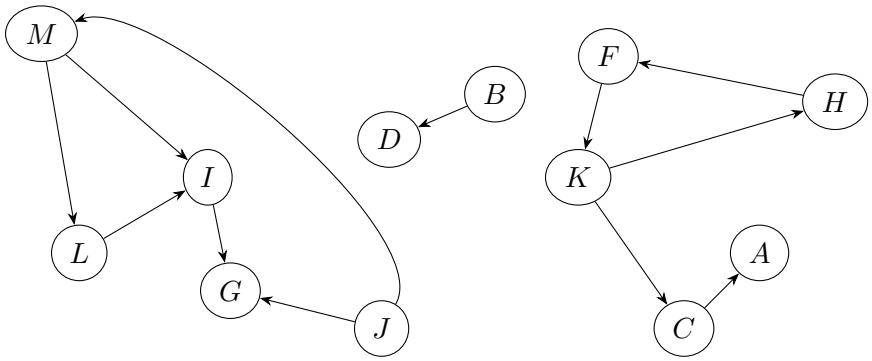

Figure 7: Disconnected graph G_C

In G_C, the components are $V_1 = \{M, I, G, J, L\}$, $V_2 = \{D, B\}$ and $V_3 = \{F, K, H, C, A\}$. The union of the different components forms the actual graph $G_C = (E, V_1 \cup V_2 \cup V_3)$. At first glance, this situation seems strange because there seems to be three different graphs, but remember, a graph represents something in real life, and for example, if in this graph an edge e_{ij} means that v_i is following v_j in Twitter, it could be possible that the components are formed (the friend group).

In directed graphs, we can differentiate between two types of connections, strong and weak. We say that two connected nodes v_i and v_j are strongly connected if $\exists\rho(v_i, v_j) \wedge \rho(v_j, v_i)$, otherwise, they are weakly connected. For example, in G_C, F is strongly connected to H because $\rho(F, H) = (F, \{F, K\}, K, \{K, H\})$ and $\rho(H, F) = (H, \{H, F\}, F)$ and B is weakly connected to D as $\rho(D, B)$ does not exist. With this definition in mind, we say that $V_s \subseteq G$ is a strongly

connected component if all $v_i \in V_s$ are strongly connected between them, for example, the subset $\{F, K, H\}$ of G_C. Note that we can not add C to the component, as C is weakly connected to K.

2.4.2 Complete graphs

A complete graph is a simple undirected graph that has all possible edges between every distinct pair of vertices $\exists e_{ij}\, \forall v_i, v_j \in V$. A complete graph with n vertices is designed as K_n. Knowing the definition, we can proceed to evaluate some of the most important properties of these kind of graphs as they will be used in the following sections.

Property 1: In a complete graph K_n:

$$|E| = \binom{n}{2}$$

Proof: A complete graph implies that every vertex is connected to all other $n-1$ vertices of the graph. This would yield us with $(n-1)n$ total edges. However, with this method, every edge has been counted two time because we have counted v_i connected to v_j and v_j connected to v_i. Hence, the number of edges in a complete graph is $\frac{(n-1)n}{2} = \binom{n}{2}$.

Property 2: The number of cycles in a complete graph is given by

$$\#Cycles = \sum_{i=3}^{n} \binom{n}{i} \frac{(i-1)!}{2}$$

Proof: Given a graph K_n, we define C_i be a cycle of length i. Now let $v_1, v_2, \ldots, v_i$ be the i vertices of this cycle. Note that for $2 < i \leq n$ there will be $\binom{n}{i}$ different combinations of elements. For each combination, we fix the first element and permute all remaining vertices in the sequence. Since we have $i-1$ remaining vertices, there will be $(i-1)!$ permutations.[7]. This number is a little bit tricky, as we do not want to count $v_a, v_b, \ldots, v_i$ and $v_i, \ldots, v_b, v_a$ as different cycles, thus,

[7] $P_k = k!$,being P_k number of unique permutations of k elements.

we have to divide it by two, hence, there are $\frac{(i-1)!}{2}$ unique cycles for every combination. Finally, we just put both observations together, so $C_i = \binom{n}{i}\frac{(i-1)!}{2}$ and after adding up all C_i we obtain that $C = C_3 + C_4 \ldots + C_{n-1} + C_n = \sum_{i=3}^{n}\binom{n}{i}\frac{(i-1)!}{2}$.

2.4.3 Cyclic graphs

We say that a graph is cyclic if it contains a cycle. A cycle is a path of length greater than 2 vertices where the last vertex is connected to the first one. $\rho(v_i, v_j)$ such that $e_{ij} \in E$. In other words, there is a cycle if we can find a sequence of vertices starting from v_i that returns to the same vertex (v_i).

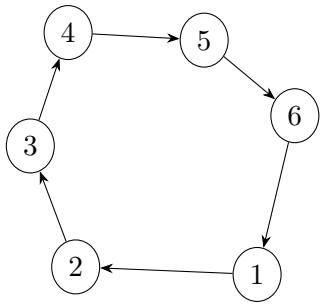

Figure 8: A cycle

2.4.4 Trees

Trees, are connected undirected graphs which do not contain a cycle (acyclic). This is a very important type of graph as it has some unique properties which I will prove in this section. Moreover, trees are not only used in graph theory, but can also be used to create really interesting data structures such as Segment Trees, Tries, binary trees, priority queues....

Propery 1: *$G = (V, E)$ is a tree if and only if $\exists!\rho(v_i, v_j)\forall v_i, v_j \in V$.*

Proof: To prove the first property and a really important one, we will do it by contradiction. Let G be a tree of n vertices. Now, suppose that there exist more than one path $\rho(v_i, v_j)$. If this was the case, we could create the cycle $\rho(v_i, v_j) \cup \rho(v_j, v_i)$, hence, if there are more than one path

from two vertices, G is not a tree.

Property 2: *If an edge is added to a tree, a cycle is created.*

Proof: Let e_{ij} be the added edge, then, as there will exist one path $\rho(v_k, v_i)$ and $\rho(v_k, v_j) \forall v_k \in V$ (property 1), if there is a connection v_i—v_j, we will have the cycle $C = v_k, \ldots, v_i, v_j \ldots, v_k$.

Property 3: *Let G be a simple connected undirected graph where $|V| = n$ and $|E| = m$, then, G is a tree if and only if $m = n - 1$.*

Proof: We will proceed by induction. The base case is trivial, as in a graph with 1 vertex, there are 0 edges, so the property holds. Assume the property to be true for all trees. Now, let $G = (V, E)$ be a tree with n vertices and m edges. When given any edge e_k that connects v_i—v_j, since e_k is the unique path $\rho(v_i, v_j)$, deleting it will create the disconnected graph $G - e_k$ with two components $G_1, G_2 \subset G - e_k$ (with n_1, n_2 vertices each) that, since there were no cycles before, both are trees. Therefore, by induction hypothesis, $m_1 = n_1 - 1$ and $m_2 = n_2 - 1$.
It can be seen that $n_1 + n_2 = n$ and $m_1 + m_2 + 1 = m$ because the number of vertices is conserved and we have deleted one edge. Substituting the first equations into this last one, it yields that $(n_1 - 1) + (n_2 - 1) + 1 = (n_1 + n_2 - 1) = n - 1 = m$ therefore, the property has been proved.

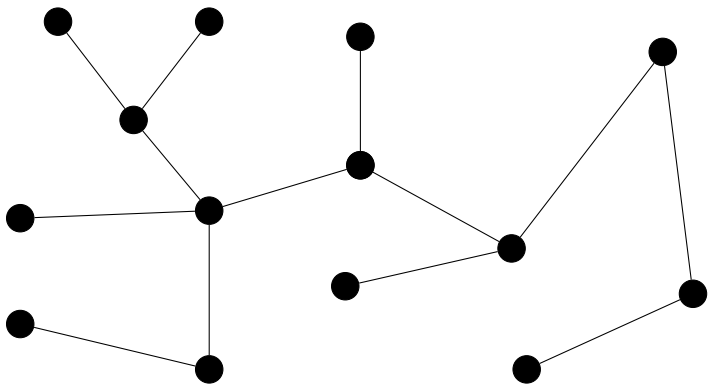

Figure 9: G_D,representation of a tree

A more intuitive way to see the proof for this last property would be starting with a tree of one

vertex and trying to add a second one. Since adding two edges together with the vertex will create a cycle, for each added vertex the number of edges will increase by one, therefore, if it holds for one vertex, which it does, it will hold for all any constructed graphs by this process.

2.5 The shortest path problem

In the following sections we will take a look on the main two algorithms to find the shortest path from a node to another. Given a path $\rho(v_i, v_j) = \{v_1, e_1, v_2 \dots, e_{m-1}, v_m\}$, we define the cost P of such path as:

$$P = \sum_{e_k \in \rho(v_i, v_j)} w(i)$$

For weighted graphs, it is the sum of the weight of all edges that have been traversed, however, in unweighed graphs, since every edges has a constant weight 1, it is much easier to count the cost of a path, as it will always be m, where m is the number of traversed edges. This simple observation will make searching the shortest path in an unweighted graph much more efficient and intuitive than in a weighted one.

Having said so, we say that $\rho_a(v_i, v_j)$ is the shortest path from v_i to v_j if:

$$\nexists \rho_g(v_i, v_j) \text{ such that } \sum_{e_k \in \rho_g(v_i, v_j)} w(i) < \sum_{e_k \in \rho_a(v_i, v_j)} w(i)$$

By the previous observation it can be said that the shortest path in an unweighted graph is the path that traversed the least eves, on the other hand, this does not hold true for weighted graphs.

2.5.1 Breadth's First Search, finding the shortest path in an unweighed graph

The BFS algorithm, also called Breadth's First Search, is the fastest way to find the shortest path in unweighted graphs. It was first designed by the mathematician Moore, and first described in *Proceedings of the International Symposium on the Theory of Switching, Harvard University Press, pp. 285–292*[AAA59]. The first purpose of the algorithm, just like the DFS was to solve complex mazes efficiently.

This algorithm will visit all vertices in an ordered matter. Starting from the source v_1, it will first visit all vertices with distance 1, which are the adjacent ones. Then, it will go to all vertices with distance $2, 3 \dots$ until all connected vertices to the source have been visited. Generalising,

if the maximum distance of the already visited vertices is x, the algorithm will visit all vertices that are adjacent to vertices with x distance. This reduces to all vertices with $x+1$ distance from the source.[8]

Initialization:

To keep track of the distance from the source, we have to initialize the array $dist[j] = \infty \; \forall v_j \in V$. Not only will this array store the shortest path from the source to v_j, but will also be used to know if a node has already been evaluated. If during the traversal we find a node v_k where $dist[k] = \infty$, it means that it has not been visited previously. Moreover, if at the end of the algorithm $dist[k] = \infty$, it means that the node v_k is not reachable form the source.

Secondly, to help us implement the retrieval of the path, we initialize a vector par, where $par[j] = i$ means that during the traversal we have taken the edge e_{ij}. This also means that, because of the way we traverse the graph, $dist[j] = dist[i] + 1$. At the end of the algorithm, if the target vertex is v_n, the path will be:

$$\{v_n, \dots par[par[1]], par[1], 1\}$$

Finally, we have to initialize a data structure called *queue* that we will call *pending* and will ensure that we always evaluate the vertices with the lowest possible distance to the source first. A queue holds a set of elements in the order they have been inserted to the queue. It has two main functions, *retrieve*, which will retrieve the oldest member of the queue and *insert* which will insert a new element in the top of the queue. *pending* will start with one element, the source and during the algorithm, vertices will be constantly inserted and retrieved from it. An example of queue in the middle of the BFS would be:

$$\text{pending} = \{\underbrace{v_a, \dots v_b}_{\text{dist}=x} \underbrace{v_c \dots, v_n}_{\text{dist}=x+1}\}$$

[8] This method will ensure that all vertices with distance x of will be visited before any vertex with distance $x+t$, $t \in \mathbb{N}^+$

It can be seen that we are currently evaluating the vertices v_i where dist[i]= x while, at the same time, inserting all their adjacent vertices which will have a distance of $x+1$. In this same example, we would now proceed to delete v_a and insert all adjacent not visited vertices of v_a to *pending*.

Procedure:

Breadth's First Search will run in a loop until *pending* is empty. In each iteration we *retrieve* a vertex v_i from the queue, then, we iterate through all v_j, adjacent to v_i and if $dist[j] \neq \infty$(This is important because if the vertex v_j has already been reached, $dist[j] \leq dist[i]+1$, so we would update with the wrong distance) we update $dist[j] = dist[i]+1$ and *insert* v_j to *pending*. At the end of the algorithm, we will have the array $dist = \{c_1, c_2, \dots c_n\}\}$ where c_i is the minimum traversed edges from the source to v_i.

Algorithm 3 BFS

Input: adj$\leftarrow \{e_1, e_2, \dots, e_{m-1}, e_m\}$

```
Declare: pending, dist,par
while pending not empty do
    v_i ← pending.retrive()
    for v_j in adj[v_i] do
        if dist[v_j] = ∞ then
            dist[v_j]←dist[v_i] + 1
            pending.insert(v_j)
            par[v_j] ← v_i
        end if
    end for
end while
```

Output: dist= $\{d_1, d_2, d_3 \dots, d_n\}$ where d_i is the minimum number of traversed edges from the source.

To retrive the path we just have to implement a basic iteration that gets the complete sequence mentioned above from *par*. In this implementation, we will call the source s and the ending vertex v_n. Furthermore, if we are only interested in the shortest path between s and an arbitrary

vertex t, we can stop the algorithm just when t is going to be inserted to the queue.

Algorithm 4 path retrieval

Input: path

Declare: path

$v \leftarrow \text{par}[v_n]$

path$\leftarrow$ path $\cup\, v_n$

while $v \neq s$ **do**

$v \leftarrow \text{par}[v]$

path$\leftarrow$ path $\cup\, v$

end while

Reverse path

Output: path$= \{s, \ldots, v_n\}$

2.5.2 Dijkstra's algorithm, finding the shortest path

Even though BFS seems a really useful path finding algorithm, it only works for unweighted graphs, which are scarce in real life. To generalize the shortest path problem to weighted graphs, we will use a very well know algorithm called Dijkstra's Algorithm. It was first conceived by the computer scientist Edsger. W. Dijkstra in 1956 and finally published in *A Note on Two Problems in Connexion with Graphs*[D$^+$59].

Declarations and process:

Let's create an array $dist[i]$ which stores the length of the known shortest path from the source to v_i in the specific time of the traversal. For example, let s be the source of the traversal. At this point we have traversed the edges $s \to v_4$ and $s \to v_2$. At this specific time, we do not know any other path $\rho(s, v_4)$ than $\{s, e_{s4}, v_4\}$, because of this, $dist[4] = w(e_{s4})$. By definition, since at the start of the algorithm we do not know any path from s to any other vertex, $dist[i] = \infty$

We will also declare the same array *par* that in BFS and DFS, however, it will be updated differently

An array $mark[i]$ will also be declared, which will store whether a vertex has been processed or not. At the start, since we have not processed any vertex, $mark[i] = false$.

The algorithm will run in $|V|$ iterations. In each iteration we will select the vertex v_i which has the minimum value of $dist[i]$ and mark it as *true*. For every chosen vertex, we will go to all non-marked vertices v_j adjacent to them, and for each exiting edge $e_{k_{ij}}$[9] we will perform an operation called **relaxation**. The operation is the following[10]:

$$dist[j] := \mathbf{min}(dist[j], dist[i] + w(e_{k_{ij}}))$$

In each relaxation, we are trying to improve the value of $dist[i]$. The explanation of a relaxation is really simple, if $dist[j] = x$, it means that we have already found a path form s to v_j with a total cost of x. If we are now evaluating the v_i and $dist[i] = y$, and we "found" an edge e_{ij}, we can theoretically take shortest path to v_i and add $\{e_{ij}, v_j\}$ which yields us a path of cost $y + w(e_{ij})$. If the cost of such path is smaller than x, we have found a better solution.

[9] As there could exists several edges e_{ij}, we differentiate them with k.

[10] The operator $a := b$ means that we are assigned the value of b to a

Algorithm 5 Naive Dijkstra's Algorithm

Input: adj$\leftarrow \{e_1, e_2, \ldots, e_{m-1}, e_m\}$, s

Declare: mark[ ]=$false$, dist[ ]= ∞

d[s]$\leftarrow 0$

for i **in** 1 **to** n **do**

$v_i \leftarrow -1$ ▷ Declare v_i, which will store the minimal

for v_j **in** 1 **to** n **do**

if (dist[v_j]<dist[v_i] **and not** marked[v_j]) **or** $v_i = -1$ **then**

$v_i \leftarrow v_j$

end if

end for

if dist[v_i]= ∞ **or** $v_i = -1$ **then**

break

end if

mark[v_i]$\leftarrow true$

for e_{ij} **in** adj[v_i] **do**

if dist[v_i]+$w(e_{ij})$ <d[v_j] **then** ▷ Performing a **relaxation**

dist[v_j] = dist[v_i]+$w(e_{ij})$

par[v_j] = v_i

end if

end for

end for

Output: $dist = \{d_1, d_2, d_3 \ldots, d_n\}$ where d_i is the sum of weights of the shortest path from the source to v_i.

Expansion: To retrieve the path, call Algorithm 4

Complexity evaluation:

To evaluate the complexity, let $n = |V|$ and $m = |E|$. In the worst case scenario the first iteration will complete itself, so it will do n iterations. This is not it, because inside every of these iterations it will have to iterate again n times to find the smallest $dist$ and k_i more, so

$n + k_i$ times. k_i is the number of adjacent edges of the selected vertex in the i^{th} iteration. It can also be seen that $\sum_{i=1}^{n} k_i = m$. This is intuitive because since every vertex is evaluated one time, and every directed edge is only adjacent to one vertex[11], it will only be traversed once (during the evaluation of its adjacent vertex). Hence, if all vertices are visited, all edges will also have been visited. Having said so we can define $T(n, k)$ as:

$$\begin{aligned} T(n,k) &= (n+k_1) + (n+k_2) \ldots + (n+k_n) \\ &= \sum_{i=1}^{n} (n+k_i) \\ &= \sum_{i=1}^{n} n + \sum_{i=1}^{n} k_i \\ &= n^2 + m \end{aligned} \tag{2}$$

Therefore, the algorithm's run time is defined by $T(n, m) = n^2 + m$ which is equivalent to $O(n^2 + m)$. We can not take m as negligible because it can be a multi-graph with $m > n^2$. This algorithm will be the most efficient when $m \approx n^2$, in other cases, such as sparse graphs, we can optimise it to run in log linear time.

2.5.3 Optimizing Dijkstra's algorithm for sparse graphs

Taking a look at the evaluation of the complexity, we realise that we take $O(n)$ to find the smallest $dist[\,]$ and $O(m)$ to perform all relaxations, ans since we do it $O(n)$ times, it yields us an $O(n^2 + m)$. When using the algorithm in a sparse graph, since m is much smaller than n^2, we ask ourselves if the first time complexity $O(n)$ could be improved without greatly affecting the relaxation time $O(m)$. Ans this is indeed, possible.

To do so, we will use a data structure called priority queue which is an array that maintains its elements sorted. It has two main functionalities, *insert* and *retrieve*. In a queue, we have seen that both methods perform in $O(1)$, but in this case, since it has to find the proper "spot" for the newly inserted element[12] both the retrieval and insertion run in $O(\log(n))$.

[11] A directed edge e_{ij} is adjacent to v_i and incident to v_j

[12] remember: It has to maintain all of its elements sorted

Having defined what a priority queue is, lets now move on the optimization. The idea is that instead of a keeping track of the distances in an array, we will use the priority queue, which will reduce the first operation from $O(n)$ to $O(log(n))$, but, will ad a $log(n)$ to relaxation process. The second part will be slower, but in total, it will be much faster. To implement this, we will take a similar approach as the BFS. To know which vertex corresponds to every distance retrieved from the priority queue, we will insert a pair of elements, $\{d, v\}$, the distance, which will be used to "sort" the queue and v, the corresponding vertex.
Since we know that when we find an element with $dist[i] = \infty$, we will only insert vertices v_j such has $dist[j] \neq \infty$. At this start, the only vertex that complies this condition is s, so we start by inserting $\{0, s\}$ to the queue. Now we now run a loop until the queue is empty, which had the queue been completely filled, we would have retrieved an element with $dist[i] = \infty$.

At each iteration of the loop, we retrieve an element from the queue. Since we can not change values inside the queue, the retrieved element could be an old element[13], meaning that let $\{d, v\}$ bee the retrieved old pair, then $d \neq dist[v]$. If we retrieve such element, we just skip it, which will not add time to the algorithm. Let $\{d, v_i\}$ the the retrieved element. We iterate though all adjacent vertices v_j of v_i and perform the relaxation. If it completes successfully, meaning that $d + w(e_{ij}) < dist[j]$, we update $dist[j] := d + w(e_{ij})$ and $par[j] := i$ then, we insert $\{d + w(e_{ij}), j\}$ to the queue.

This new way to approach the problem is much faster and does not require the mark array. As this will affect both operations, we will get a final complexity of $O(nlog(n) + mlog(n)$ which at its turn, as it is a sparse graph, reduces to $O(mlog(n))$.

[13]The element could have been inserted and before being evaluated inserted again with an even shorter distance

Algorithm 6 Optimized Dijkstra's algorithm

Input: adj$\leftarrow \{e_1, e_2, \ldots, e_{m-1}, e_m\}$

Declare: priorityQueue *pending*, dist[ ]$= \infty$

while *pending* **not** empty **do**

$\{d, v_i\} \leftarrow$ pending.$retrieve()$

for v_j **in** adj$[v_i]$ **do**

if $w(e_{ij}) + d < dist[v_j]$ **then**

$dist[v_j] \leftarrow w(e_{ij}) + dist[v_j]$

pending.$insert(\{dist[v_j], v_j\})$

$par[v_j] = v_i$

end if

end for

end while

Output: $dist = \{d_1, d_2, d_3 \ldots, d_n\}$ where d_i is the sum of weights of the shortest path from the source to v_i.

Expansion: To retrieve the path, call Algorithm 4

2.6 Graphs to model real life situations

This last section of graphs theory aims to synthesize what is needed to apply graphs to a real world situation. To define a graph, we only need two things, the graph itself (edges, vertices and edge cost) and the meaning of each edge. The first one is the way in which the graph will be interpreted and the second one the graph configuration (nodes, edges, edges cost)[14].

2.6.1 Interpretation of the graph

The interpretation of a graph is a one way process. At first a problem to solve has to be found and after, graph theory can be used to model the situation and maybe answer the problem. To interpret a graph two things have to be defined, the meaning of an edge and the meaning of a vertex. Then, the property of the graph that gives us the answer has to be found. Finally,

[14]This last one is dependant on the interpretation of the graph

an algorithm has to be designed to find this property. To further on this, an example is given: Network modeling.

Imagine that it is desired to approximate the friend groups of the students of a school[15]. Graph theory can be used to do that. Each individual of the school is going to represent a node in our graph[16]. Then, two nodes are going to be connected by a directed edge if the student A follows the student B in social media[17]. At the end, the friend groups will be the components of the graph[18]. If a component is really dense, the friend group can be said to be solid, otherwise, maybe there are some internal discrepancies. At the end, an algorithm has to be designed to find to components of a graph which will be a modified DFS that I have already explained.[19].

2.6.2 Construction of a graph

Constructing a graph, after the interpretation of the graph is clear. We only need three things. A dataset A that gives us all the possible edges of the graph, a dataset B that gives us all the possible connections of the graph and a dataset C that gives us the weights of the graph. In our example case, thee edges have to weight, nevertheless, had we interpreted each edge as a road, we would need an experimentally created dataset of the distances of each road.

$$A = \begin{bmatrix} v_1 & v_2 & \dots & v_n \\ name1 & name2 & \dots & name16 \end{bmatrix} B = \begin{bmatrix} name1 & name2 & \dots & name16 \\ \downarrow & \downarrow & \vdots & \downarrow \\ name16 & name16 & \dots & name2 \end{bmatrix} \tag{3}$$

$$C = \begin{bmatrix} w_1 & w_2 & \dots & w_n \end{bmatrix} \tag{4}$$

Once this datasets are given, we get the adjacency list from them and run the algorithm designed in the last point to get the properties we are interested in.

[15]The problem is defined

[16]Nodes meaning is defined

[17]For clarification, he edge e_{ij} means that i follows j in social media

[18]The answer is found

[19]Last step, algorithm design

"Innovation is taking two things that exist and putting them together in a new way."

Tom Freston

This page has been intentionally left blank

3 Non-static graphs, definitions and theoretical analysis

3.1 Graph limitations

Traditional graphs have a detrimental flaw; once defined, they do not interact with the algorithm that runs on them and the weights of the edges remain constant through all the process. This leads to some major difficulties when trying to model some complex situations.

3.2 A limited problem example, Shortest path

To solve for the shortest path, just like it has been done in section 2.5, Dijkstra's algorithm is executed on a weighted positive cost graph. A common real world application would be finding the shortest path from my house, A and my best friends house B. To find it, each point of interest is represented with a vertex and each road between points of interest are edges. The intuitive approach in this case would be defining the cost of each edge as the length of the road that represents. However, this method would yield us the shortest path in distance and not in time. To get the time, considering a constant velocity over each edge[20], time can be expressed as a function of x, the length of the road and the maximum velocity allowed:

$$t = \frac{x}{v}$$

Executing the Dijkstra's on the same graph but were the weight of each edge is $\frac{x}{v}$ would make the result even better.

Even if this method seems accurate enough, it does not take into account traffic. One thing about traffic is that it changes over time, so defining the cost of the edge at the start of the algorithm does not work. To solve it, the graph would have to be deleted and redefined again for every edge that is traversed which takes insane amount of memory and computational time, which makes the solution useless for the small amount of imprecision it tries to solve. This is a really simple example, amongst many others, on when graphs fail to properly abstractly describe a real world situation. This is why it all points out to a new type of graph non-static graphs.

[20]The velocity over each edge is the maximum velocity that is allowed for cars in that road

3.3 The generalized concept

To proceed with the explanation we need to first define some important new concepts. A non-static graph $\hat{G} = (\hat{E}, V)$ of size n will be the ordered union of all infinitesimally small static graphs G of size n, being small the span of time in which their configuration exists.[21] A non-static graph does not have a defined number of edges, each configuration can have different edges, but all of them have the same number of nodes[22]

$$\hat{G} = \{G_1, G_2, G_3 \ldots, G_\infty\}$$

Consider G_t to be a concrete configuration in the span of time t of $\hat{G}$, to access it we will interpret it as a function and write $\hat{G}(t) \to G_t = (E_t, V)$. To access the concrete edge set $\hat{E}(t) \to E_t = \{e_1, e_2, e_3 \ldots, e_k\}$ where $e_i = \{w_i, s_i, t_i\}$. Note that for any given t, the concrete configuration will be different. Each concrete configuration behaves exactly as a normal graph, so we can access the source with $s(e_i)$ and the target with $t(e_i)$ and the weight with $w(e_{ij})$. To know how $\hat{G}$ evolves over the variable t, each edge of the $\binom{n}{2}$ edges of the complete graph will have a function $f_{ij}(t)$ assigned, which will yield the cost of that edges in the span t. To complement this, each non-static graph has its own error function which decides whether an edge is existent or not in a concrete time.

3.4 Deleting edges, the error function

Right now we have defined a graph which has constant edges but the cost of them changes over a value t. This is quite better than a static graph, but how do we know whether an edge e_{ij} is existent in a concrete configuration G_t? To do so, the boolean function $error(e_{ij})$ has to the defined. This function takes as parameters an edge of the graph and return 0 if the edge exists or 1 if the edge does not exist. To evaluate the existence of an edge, we consider a condition. For example, a rather elemental condition could be considering a boundary ϵ and defining the error

[21] Note that the vertex set does not have the hat which marks interactivity since it will remain constant through all other configurations.

[22] Nodes can also be weighted and interpreted non-statically, but for now we will define them as constant, because what is a node if is has any incident nor adjacent edges.

function as follows:

$$error(e_{ij}, t) = \begin{cases} 1 \text{ if } f_{ij}(t) < \epsilon \\ 0 \text{ if } f_{ij}(t) \geq \epsilon \end{cases} \tag{5}$$

These exact error function is the one that is going to be considered in the next section in all the figures that represent a non-static graph. If the interpretation of the graph makes less important edges with high weights, it makes sense to consider this function, but had it been another case, this error function would not make a lot of sense.

It is in this moment where we can start to see that non-static graphs are very dependant on the situation they represent.

To make this example error function more interesting, epsilon can be interpreted as a function and put as condition $f_{ij}(t) < \epsilon(t)$. In fact, the error condition can be set in any way it is desired if it returns 0 or 1 without overlapping, meaning that the condition must be or True or False for any given inputs. Over this article, many varieties of error functions are used, but to make it clearer here is another example of error function:

Let S_{ij} be the set of all adjacent edges of v_i and v_j, then, a possible error function would be:

$$error(e_{ij}, t) = \begin{cases} 1 \text{ if } |\sum_{e_k \in S_{ij}} w(e_k)| \geq \epsilon t^2 \\ 0 \text{ if } |\sum_{e_k \in S_{ij}} w(e_k)| < \epsilon t^2 \end{cases} \tag{6}$$

In this error function, an edge e_{ij} exists if the sum of all weights of the adjacent vertices of v_i and v_j is greater that the value of ϵt^2. Taking this occasion, I would like to point out that for any configuration $\hat{G}(t)$, the generator function create a complete graph and the error function interprets this complete graph and deletes the corresponding edges. This means that it is not mandatory to defined one, everything will depend on the situation. It is possible that just with the changing cost of the edges it is enough[23].

[23]It could be possible to use the costs of each edge only as a way to determine whether an edge exists and use the resulting graph as a non-weighted graph

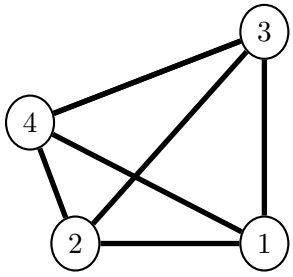

Figure 10: Example of $\hat{G}(t)$

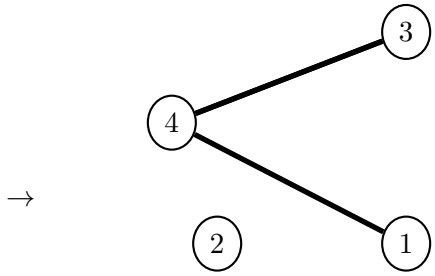

Figure 11: Possible result after going through the error function

3.5 Representation of a non-static graph

Now that we have defined a way to mathematically reference and represent a non-static graph the way to draw it on paper has to be defined. To do so, we will represent the complete graph K_n with dotted edges. For specific configurations of the graph, the existent edges will be represented with black lines and the ones that have been deleted by the error function will be left as dotted lines. To make an edge always in existent we do just like a normal graph, setting the generator function to a constant such as ∞ which will for sure always make the error function evaluate it as 0[24]. In this cases, the dotted edge can be skipped.

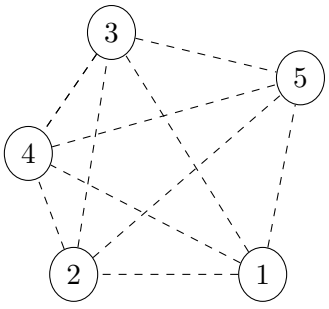

Figure 12: Non-static graph $\hat{G}$

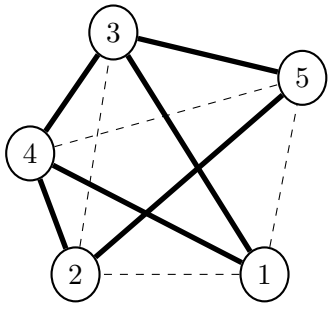

Figure 13: Arbitrary configuration in time t_0

3.5.1 Adjacency matrix and concrete configurations

Each directed edge e_{ij} will have one corresponding function $f_{ij}(t)$ which will give the cost of the edge e_{ij} in an instant t and every undirected edge e_k will have one function $f_k(x)$ which will

[24]Another solution would be: To delete e_k, an extra condition on the error function can be added: 0 if $e_i = e_k$

work in the two directions of the edge. These functions are stored in an $n \times n$ adjacency matrix, where instead of the weights, there is a function. Moreover, the adjacency matrix of a concrete configuration t_0 is just $ADJ(t_0)$.

$$ADJ(t) = \begin{bmatrix} \infty & f_{12}(t) & f_{13}(t) & f_{14}(t) & f_{15}(t) \\ f_{21}(t) & \infty & f_{23}(t) & f_{24}(t) & f_{25}(t) \\ f_{31}(t) & f_{32}(t) & \infty & f_{34}(t) & f_{35}(t) \\ f_{41}(t) & f_{42}(t) & f_{43}(t) & \infty & f_{45}(t) \\ f_{51}(t) & f_{52}(t) & f_{53}(t) & f_{54}(t) & \infty \end{bmatrix} \tag{7}$$

Figure 14: Non-static adjacency matrix

3.5.2 Three dimensional representation of a non-static graph

Since the graph is defined by some functions, to visualize it better, we can represent it in a three dimensional Cartesian plane of coordinates. Putting a graph in a two dimensional space is relatively easy, we just have to give each node a coordinate, and draw the corresponding graph. To add the third dimension we will use t, which will be the z axis and x, y will be used to represent each specific configuration of the graph. To really see the evolution of the graph on the z axis, every representation will be defined by a range of value $[a, b]$, and increment k and obviously a graph $\hat{G}$. Each specific configuration will be drawn in the chosen range in a separation of k.

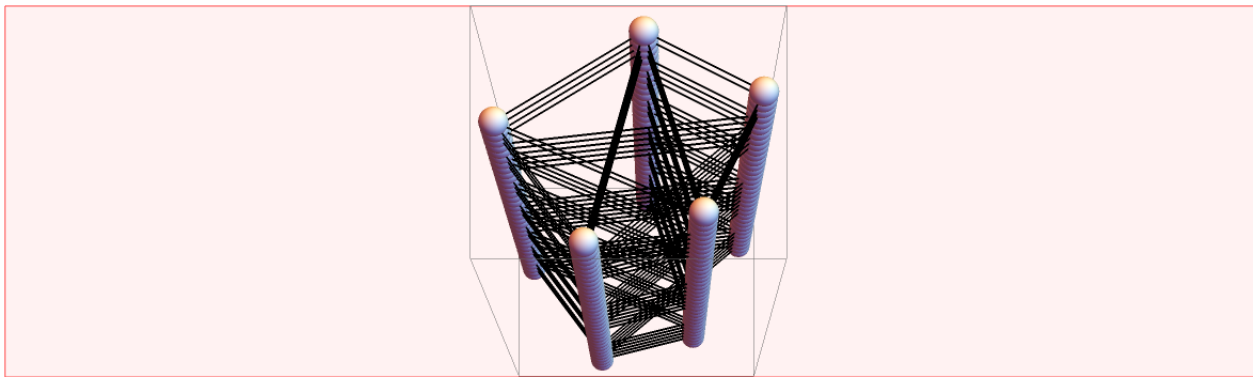

Figure 15: $\hat{G}_1$

3.5.3 Considering arbitrary generator functions as an example

To keep with the explanation, two examples of arbitrarily chosen graphs are going to be presented. To see some differences, one graph will be made of trigonometric functions and one out of polynomials of low degree.

In this first example, as we can see in figure16 five polynomial function have been taken as generator functions. It can clearly be seen that the graph, as it should, follows a similar behaviour as the functions.

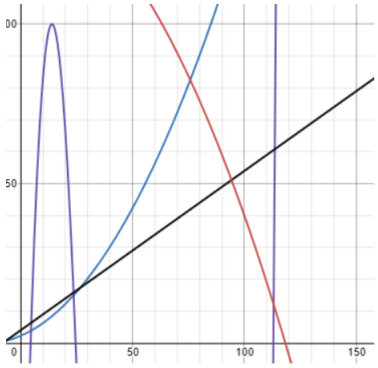

Figure 16: Representation of the functions of the polynomial graph

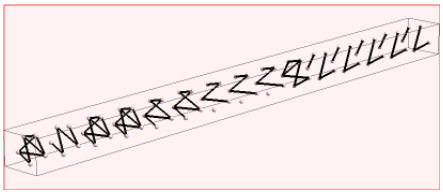

Figure 17: Polynomial graph

On the other hand, in figure18 graph a mix of diferent trigonometric functions have been taken which as it can be expected make a rapidly changing graph.

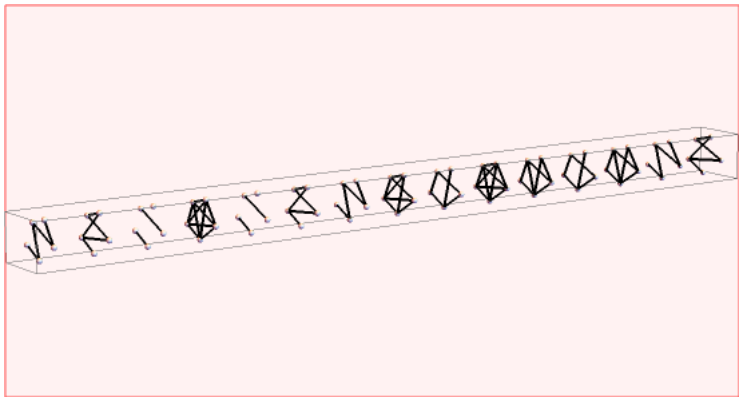

Figure 18: Trigonometric graph

3.6 Non-static trees, an error function example

Trees are a really useful type of graph, it allows us, for example, to compute the most efficient way to connect n cities (MST [Kru56] algorithm). We have seen that a connected graph is a tree if and only if it has n–1 edges, being n the number of vertices of the graph. In non-static graphs, we say a graph $\hat{G}$ is a tree in the interval $[a, b]$ if and only if $\hat{G}(t)$ has n–1 edges for all $t \in [a, b]$ and each configuration in that interval is connected. If it holds for all t, we say the graph is a pure tree. However, it is hard to find functions such that this condition holds. The easy solution would be defining $n-1$ edges with functions, set the other edges to ∞ and not consider ϵ. Even though doing this might seem an Occam razor, there is a much cleaner way to do it.

Given a concrete configuration G_k of a graph $\hat{G}$, let T_i be a tree, then we define the minimum spanning tree $T_1 = MST(G_k)$ as a connected subgraph $T_1 \subset G_k$ such that $\nexists T_2 \subset G_k$ such that:

$$\sum_{e_i \in T_2} w(e_i) < \sum_{e_j \in T_1} w(e_j)$$

Having said so to make a forces non-static tree the error function can be defined as follows:

$$Error(e_k, t) = \begin{cases} 1 \text{ if } e_k \in MST(G_t) \\ 0 \text{ if } e_k \notin MST(G_t) \end{cases} \tag{8}$$

This error function will ensure that every concrete configuration is a static tree. Being built in a constructive matter, it is a perfect example of the freedom that the error function allows the designer of a non-static graph, thing that opens a broad horizon of possibilities and opportunities to graph theorists and computer scientists.

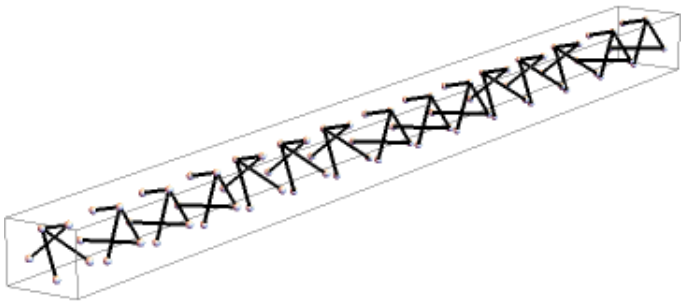

Figure 19: Tree graph built with MST

Even if it seems that the configurations shown above contain cycles, it is only an illusion since it is only the edges crossing over in a place where there is no vertex.

3.6.1 Disjoint Set Union, DSU

To construct the minimum spanning tree from a given graph, we will use a very well known algorithm called Kruskal MST that runs in a very optimized[25] $O(n\log(n))$. To explain the algorithm, knowledge about Disjoint set union is required and keeping in mind that it is related to graphs, It is going to be explained.

DSU, also known as Disjoint Set Union is a graph based data structure used amongst many other things, to tell whether adding an edge to a graph will create a cycle. DSU consists of mainly an array $par = \{1, 2, 3 \dots n\}$ of length $n = |V|$, a recursive function $find(v_i)$ and a function $add(e_i)$. In reality the DSU is a graph by itself and $par[i] = j$ means that v_i is connected to v_j. This means that each vertex can only be connected to one other vertex.
To know why it works it is essential to remember one property of a graph; If an edge e_{ij} is added to a graph and v_i and v_j belong to the same component, it will create a cycle, otherwise it will not. The proof is exactly the same as Property 2 of section 2.4.4 Tree graphs. Sticking to this property, it can be known whether adding an edge to the graph will create a cycle. Given an

[25]It works faster, but still not as fast as an $O(n)$, so it has to be noted as it is

edge e_{ij}, if the vertices v_i and v_j belong to the same component it will add a cycle.

Without entering into many detail, what the DSU does is keeping track of the components of the graph. Each component of the graph has a leading vertex which is like an id of the component. If v_i is in the component x, and x has a leading vertex v_k, the vertex v_i will be connected to v_k in the DSU graph which means that $par[i] = k$ and $par[k] = k$.

Keeping this in mind, the find function, with argument v_i gives the leading vertex of the component where v_i belongs. Then, $find(i) = find(j)$, we know that the vertices v_i and v_j are in the same component because they have the same leading vertex. Therefore, if we desire to add an edge e_{ij} and both adjacent vertices have the same leading vertex, $find(j) = find(i)$ we know we will be creating a cycle because On the other hand, the $add(e_k)$ function of the property mentioned above. The $add(j)$ function allows to add an edge the the graph. Given an edge e_k, it adds it to the DSU. This process is really optimized with recursive path compression and other things, but I will not go into further detail but the optimizations are implemented in the code.

Algorithm 7 DSU find and add

Input: par[ ]

```
procedure FIND(v_i)
    if par[v_i] = v_i then
        return v_i                          ▷ It has arrived to the leading vertex and returns it.
    else
        return par[v_i] = find(par[v_i])    ▷ Path compression
    end if
end procedure

procedure ADD(e_k)
    par[find(s(e_k))] = par[find(t(e_k))]   ▷ Joins the two leading vertices into
                                              the same component.
end procedure
```

3.6.2 Constructing the MST for each configuration

To proceed with the extraction of the MST, we first need an array of edges, $edges = \{e_1, e_2 \dots e_n\}$. If we dispose of an adjacency list or matrix, we just iterate through all of it and add each edge in a the new array *edges*, which will take $O(m)$. After this, the array is sorted in increasing order of edge weights. It iterates through all edges in the sorted array and for each edge, with the find function of DSU, it checks whether the target and the source of the edge are in the same component, if they are, it would form a cycle and skips them, it they are not, it adds them to the DSU and the MST final array. At the end, the algorithm will return an array of all edges belonging to the MST which are the edges of the concrete configuration of the non-static graph we need.

Algorithm 8 Kruskal MST

Input: Add, Find, par, edges

```
Sort(edges)
MST = {}
for e_i in edges do
    if Find(t(e_i)) ≠ Find(s(e_i) then
        add(t(e_i), s(e_i))
        MST ← MST ∪ e_i
    end if
end for
```

Output: MST

Since this algorithm is relatively fast, it is doable to implement it in our non-static graphs if we want it to be a tree. To do so, each time it is mandatory to extract a concrete configuration $\hat{G}(t_0)$, the algorithm has to be called with edge set that $\hat{E}(t_0)$ yields. However it is a rather slow approach because it will take $O(mn\log(n))$ time, being m the number of configurations we want to access.

3.6.3 Proof of the validity of Kruskal's greedy algorithm

To prove that Kruskal's algorithm works properly, we will proceed by contradiction. Let T_m be the tree that the MST algorithm has produced running on the graph G. If it is not the MST, it means that there exists $e_i \in T_c$ and $e_j \in G$ such that $w(e_j) < w(e_i)$ and if e_i is deleted from T_c and replaced by e_j, the resulting graph is still a tree. If e_i is deleted, since it is a bridge, it will produce two components. By definition of tree, e_j must connect these two components. However, if $w(e_j) < w(e_i)$ the edge e_j would have been evaluated before e_i and since it is the only vertex that joins these components it does not create a cycle which means e_j belongs to the MST, hence, we have a contradiction.

4 Non-static graphs as an abstract model

Now that non-static graphs are formally defined, it is time to explain how to use them to model real world situations. In the case of traditional graphs, to model a real world situation only two things are needed, the configuration of the graph -which edges exist and the number of vertices- and a dataset which gives us the weights of each edge. Keeping the road example, we would need the information about which roads and which points of the city they connect (edges and vertices) and the length and average speed of each road. This, in traditional graphs corresponds to the adjacency matrix of the graph. As it has already been commented in section 2.6, since normal graphs have a defined weight for each edge, it is not needed to have various values for each edge.

4.1 Required dataset

As the configuration of the graph concerns, to create non-static graphs, since we have a complete graph, it only needs the number of vertices because the number of edges will be, by definition, $\binom{n}{2}$.

When it comes to the weight of the edges, it becomes much harder to get what is needed to create the generator functions. In a traditional graph, since the weights are constant, we only need a one dimensional dataset with the weights of each edge $Dataset = \{w(e_1), w(e_2) \dots w(e_n)\}$, however, since for non-static graphs the weight is non-constant, it becomes a problem which has two solutions.

The easy way out is when the generator function of the situation that we need to model is well known and given to us. In this case, which is not typical, we just get the generator function of each edge and insert it to the adjacency matrix. The other option is to collect a similar dataset to the mentioned above. In this case, we have a dataset with one dependent variable which will be the weight of the graph and one ore more independent variables that define that cost. If we put these information inputs in a Cartesian plane of coordinates where Y is the weight of the graph, in the case of a single independent variable we get something similar to figure 20. Note that these values are experimentally collected and will not be all precise. However, it can clearly be seen that they follow a tendency which will be the generator function.

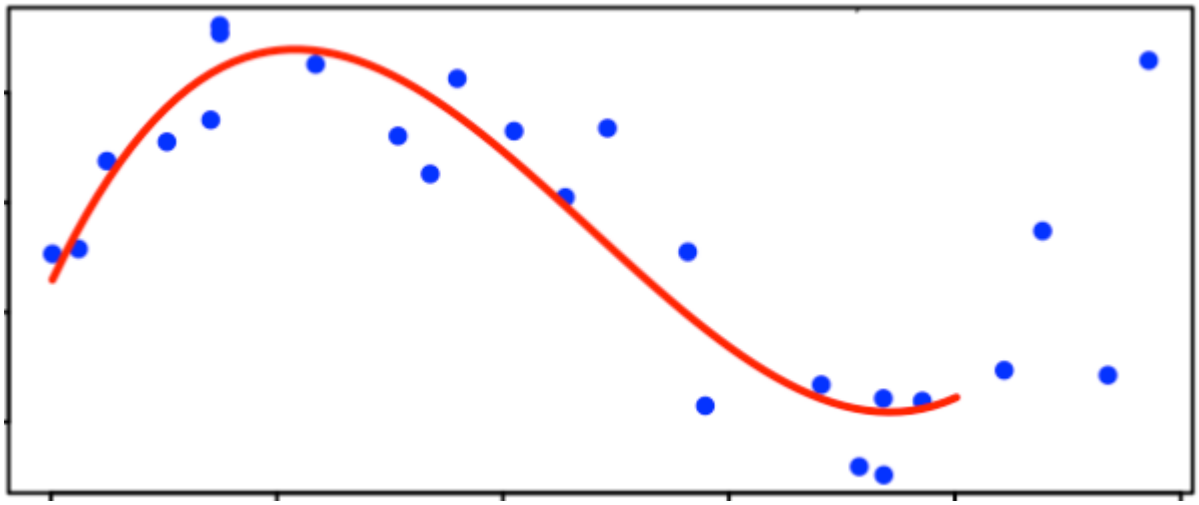

Figure 20: Example of an edge dataset and the tendency curve

This dataset that has been mentioned is the definition of a single edge dataset, however, we need this for every of the $\binom{n}{2}$ edges of the graph and note that the generator function can be dependant of several variables, thing that would leave us with an multiple dimensional graph. For the general case, lets consider that we have a graph dataset $D = \{P_1, P_2 \dots, P_{\binom{n}{2}}\}$ of pairs of $P_i = \{X_i, Y_i\}$ of experimentally collected values where X_i is an k-tuple and $Y_i \approx f(x_{i1}, x_{i2} \dots x_{ik})$. (as shown in equation 9. Our goal is to find a function $f(x_1, x_2 \dots x_m)$ for each P_i which adjust best to the dataset at our disposal. This function will be the generator function of the edge e_i as can be seen represented in the example above by the red curve. It can be clearly seen that the curve is not adjusted perfectly to all points, but it follows the tendency good enough to get accurate results. Furthermore, it would even be negative to adjust it more precisely. Let this be the case, we would have a case of over fitting.

$$X_j = \begin{bmatrix} x_{11} & x_{12} & \dots & x_{1k} \\ x_{21} & x_{22} & \dots & x_{2k} \\ \vdots & \vdots & \ddots & \vdots \\ x_{n1} & x_{n2} & \dots & x_{nk} \end{bmatrix} \qquad Y_j = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_m \end{bmatrix} \tag{9}$$

For each X_i, Y_i, which are the edge datasets of edge e_i, to find the corresponding generator function f, we have 3 cases in increasing number of complexity:

1. If $k = 1$ and the dataset follows a simple linear tendency of the form $y = ax + b$ plus an error, as will be studies in section 4.2 Simple linear tendency the generator function will be the regression straight of the dataset $f(x) = \alpha x + \beta$.

2. If there is a single dependent variable defined by m independent x_i and the dataset follows a linear m dimensional tendency $y = \alpha_1 x_1 + \alpha_2 x_2 \dots \alpha_m x_m + \beta$, the generator function will be the m dimentional regression hyperplane, $f(x_1, x_2 \dots x_m) = \alpha_1 x_1 + \alpha_2 x_2 \dots \alpha_m x_m + \beta$, which adjusts better the the dataset. See 4.3 Multi-variable linear tendency

3. In the last case, we encounter a single dependent variable defined by a single independent one which does not seem to follow any concrete tendency. For these cases, which in real world applications are a vast majority, we will use an expansion of the second case to fit a k degree polynomial to the dataset which will be found transforming the polynomial to a linear function by adding dimensions to the dataset. Explained in the section 4.4

4.2 Simple linear tendency

In this case, suppose our edge dataset $E_i = \{[x_1, x_2 \dots, x_n], [y_1, y_2 \dots y_n]\}$ follows a linear tendency of the form $y = ax + b + \epsilon$ where epsilon is a small arbitrary noise or experimental error. It is important to note that as the the average noise gets smaller, the resulting function becomes more accurate.

With our assumption in mind, we can proceed the find the generator function of that edge, which is the well known regression curve $f_i(x) = \widehat{y} = \alpha x + \beta$. To solve for the slope and for the intersection on coordinates, the minimization of error squared method will be used. In this case, this can be done with plane maths, however, in further section, some computational approximations will be needed.

In figure 28 we can see an example of what an edge dataset with a linear tendency looks like. Just beside it, I have run the linear regression algorithm to find what would be the generator function of this dataset. As it has already been said, the function does not exactly fit the dataset,

however, it adjusts considerably right and will therefore give an accurate approximation of the generator function.

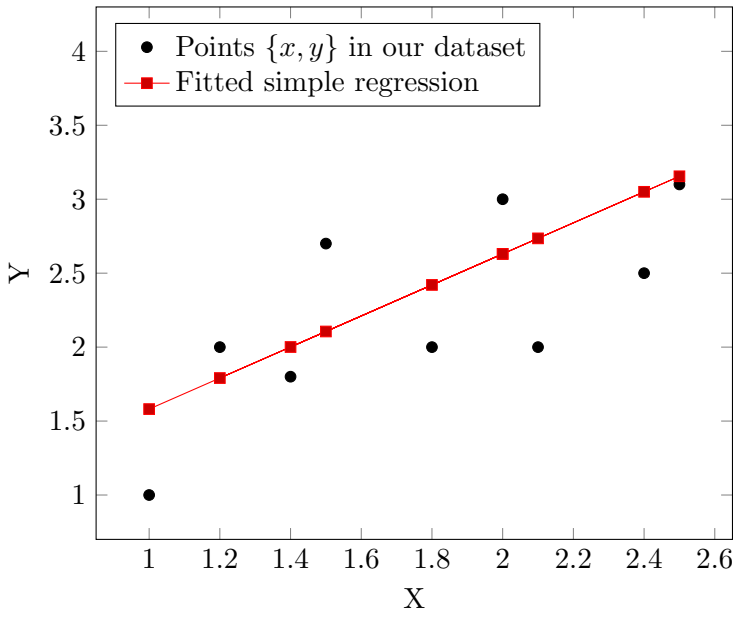

Figure 21: Plot of a linear regression

$$\begin{bmatrix} X & Y \\ 1 & 1 \\ 2 & 3 \\ 2.5 & 3.1 \\ 1.5 & 2.7 \\ 1.4 & 1.8 \\ 2 & 4 \\ 1.2 & 2 \\ 1.8 & 2 \\ 2.1 & 2 \\ 2.4 & 2.5 \end{bmatrix} \tag{10}$$

Figure 22: Dataset

4.2.1 Minimization of the error squared

The minimization of errors squared is a well known technique which consists on finding α and β such that the sum of squared errors is minimum. To proceed with that, let $\widehat{y}$ be the predicted y of a specific x_i of the dataset and e_i the corresponding error of the prediction, therefore:

$$e_i = |y_i - \widehat{y}| = |y_i - \alpha x_i - \beta|$$

Hence, to find alpha and beta, we have to find α and β such that:

$$\sum e_i^2 = \sum (y_i - \alpha x_i - \beta)^2$$

is minimum

Note that even though the "error" would be $|y - \widehat{y}|$, we are squaring it. Not only is squaring the error a mathematical convenience to have a differentiable function but also, by the behaviour of x^2 in comparison of $|x|$, it gives more emphasis to points which are further away from our

function, which in most cases results in better predictions. However this could turn out to bias our model towards out-layers, points which we will have to take care before processing our data.

To see what we are dealing with, it is a good idea to plot the function $y = f(\alpha, \beta) = \sum e_i^2$ for some specific dataset (figure 28). Since there are 3 variables, the pot will be three dimensional. Using the regression algorithm, we find that the minimum point of the function is $\alpha = 1.04936$ and $\beta = 0.531644$ which means that the adjusted regression straight is $y = 1.04936x + 0.531644$ which yields us our generator function.

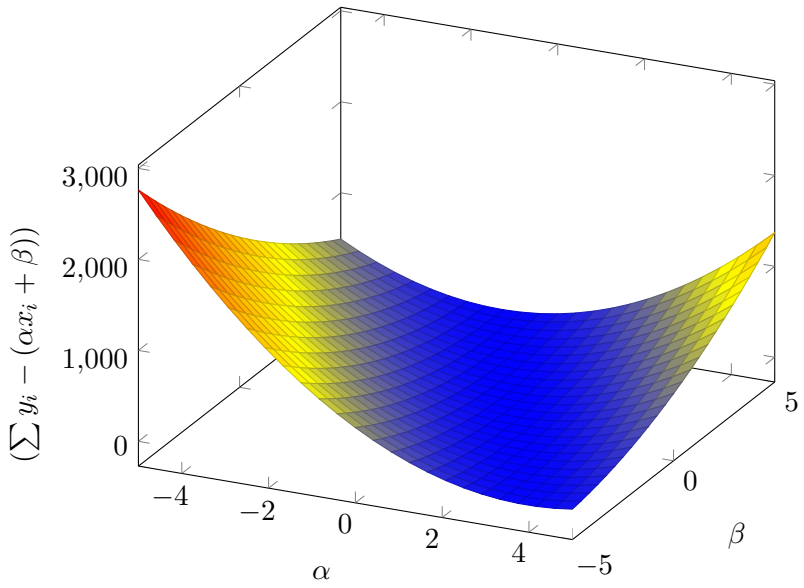

Figure 23: Plot of $y = \sum e_i^2$ of figure 28

After this clarification, lets proceed to the minimization of the sum of the squared error, which can easily be done using partial derivatives. Now let $f(\beta, \alpha)$ express the sum of errors in function of α and β, then:

$$f(\alpha, \beta) = \sum_{i=1}^{n} e_i^2 = \sum_{i=1}^{n} (y_i - \alpha x_i - \beta)^2$$

Now we just have to calculate the values of α and β such that $f(\alpha, \beta)$ is minimum, which will yield the coefficient of x and the independent therm in our equation for the regression straight line. First of all, we have to take the two possible partial derivatives and arrange the expression in such way that it sticks to the form $a\alpha + b\beta + c$. Note that x_i and y_i behave like constants,

because they are the values that are in our dataset.

$$\begin{aligned}\frac{\partial f}{\partial \alpha} &= -2\sum_{i=1}^{n}(y_i - \alpha x_i - \beta)x_i = 2\alpha\sum_{i=1}^{n}x_i^2 + 2\beta\sum_{i=1}^{n}x_i - 2\sum_{i=1}^{n}y_i x_i \\ \frac{\partial f}{\partial \beta} &= -2\sum_{i=1}^{n}(y_i - \alpha x_i - \beta) = 2\alpha\sum_{i=1}^{n}x_i + 2n\beta - 2\sum_{i=1}^{n}y_i\end{aligned} \tag{11}$$

Having done that, to find the, critical point, which, since it is a concave function, will be a minimum, we assign the formulas to 0 and solve the system of equations. To solve the system, even though it is only a two variables system, we will use matrices, because for further generalisations with m variables, this same method will be used. It is well known that a system of equations like the following can be rewritten as:

$$\begin{aligned}a_1 x + b_1 y &= c_1 \\ a_2 x + b_2 y &= c_2 \\ &\downarrow \\ \begin{bmatrix} a_1 & b_2 \\ a_2 & b_2 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} &= \begin{bmatrix} c_1 \\ c_2 \end{bmatrix}\end{aligned} \tag{12}$$

Having said so, we have to write our equations so they follow the indicated form. To do so, we divide both side by 2, and rearrange the expression a little bit. To write the algebra clearly, the first matrix will be called A, the second one B, and the third one R.

$$\begin{aligned}\alpha\sum_{i=1}^{n}x_i^2 + \beta\sum_{i=1}^{n}x_i &= \sum_{i=1}^{n}y_i x_i \\ \alpha\sum_{i=1}^{n}x_i + n\beta &= \sum_{i=1}^{n}y_i \\ \begin{bmatrix} \sum x_i^2 & \sum x_i \\ \sum x_i & n \end{bmatrix} \begin{bmatrix} \alpha \\ \beta \end{bmatrix} &= \begin{bmatrix} \sum y_i x_i \\ \sum y_i \end{bmatrix} \\ A \cdot R &= B\end{aligned} \tag{13}$$

We proceed to solve it:

$$\begin{aligned}A{\cdot}R &= B \\ R &= A^{-1} \cdot B \\ R &= \frac{1}{|A|} \cdot adj(A)^T \cdot B\end{aligned} \tag{14}$$

This yields us:

$$\begin{bmatrix}\alpha\\ \beta\end{bmatrix} = \frac{1}{n\sum x^2 - \left[\sum x_i\right]^2}\begin{bmatrix} n\sum y_i x_i - \sum x_i \sum y_i \\ \sum x_i^2 \sum y_i - \sum x_i \sum x_i y_i\end{bmatrix}$$

which at the end gives us the following expressions for α and β.

$$\alpha = \frac{n\sum y_i x_i - \sum x_i \sum y_i}{n\sum x^2 - \left[\sum x_i\right]^2} \qquad \beta = \frac{\sum x_i^2 \sum y_i - \sum x_i \sum x_i y_i}{n\sum x^2 - \left[\sum x_i\right]^2}$$

4.2.2 Implementation of the algorithm

Implementing the fit function of a Simple Linear Regression is really easy, once we have gone through the work to deduce the formula for α and β, we only have to compute all summations and use the formula to find both variables. To do so, we iterate through all the dataset and actively add the corresponding values to the summations, which we will call xSum, xsSum, xySum, ySum.

Once we have found α and β, to predict the estimated $\widehat{y}$ with a given x, we just have to calculate $\alpha x + \beta$, which I consider trivial and will not implement.

Algorithm 9 Linear Regression algorithm

Input: dataset[n][2], n **Declare**xsSum,xySum,xSum,ySum

for i **in** 1 **to** n **do**

for variable **in** $\{x, y\}$ **do**

if variable = x **then**

xsSum← xsSum + $dataset[i][1]^2$

xSum← xSum + dataset[i][1]

yxSum← yxSum + dataset[i][0]*dataset[i][1]

else

ySum← ySum + dataset[i][0]

end if

end for

end for

den← $n \cdot xsSum$ - $xSum^2$

$\alpha \leftarrow \frac{(n \cdot xySum - xSum \cdot ySum)}{den}$

$\beta \leftarrow \frac{xsSum \cdot ySum - xySum \cdot xSum}{den}$

Output: α, β

4.2.3 Resulting graph

Let $SimpleRegression()$ be the algorithm implemented above which takes as parameter an edge dataset and return a pair (α, β) of the coefficients corresponding the the regression straight. Consider now $data_{ij}$ as the edge dataset corresponding to the edge e_{ij}, therefore, the final generator function $f_{ij}(x)$ is the following:

$$\begin{aligned} a =& SimpleRegression(data_{ij})[0] \\ b =& SimpleRegression(data_{ij})[1] \\ f_{ij}(x) =& ax + b \end{aligned} \tag{15}$$

Let us finish the section with a walk through of an example on a graph that can be completely generated only from linear functions. Since finding a situation that can be interpreted with a graph that only using linear functions is a very rare case to find in a real world situation, but to

to have the dataset, we will automatically generate it as an example.

Lets consider now the following graph $\hat{G}$ with 5 vertices. Suppose we have an edge dataset for each edge that seems to follow a simple linear tendency. In this case, as we have seen, linear regression can be used to find the generator function. [26]

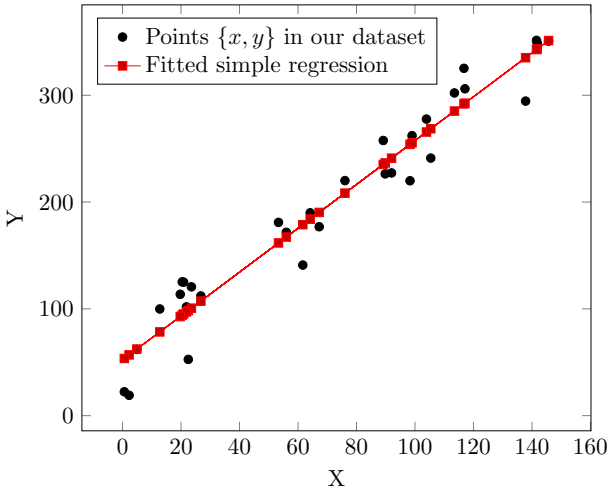

Figure 24: edge dataset 3

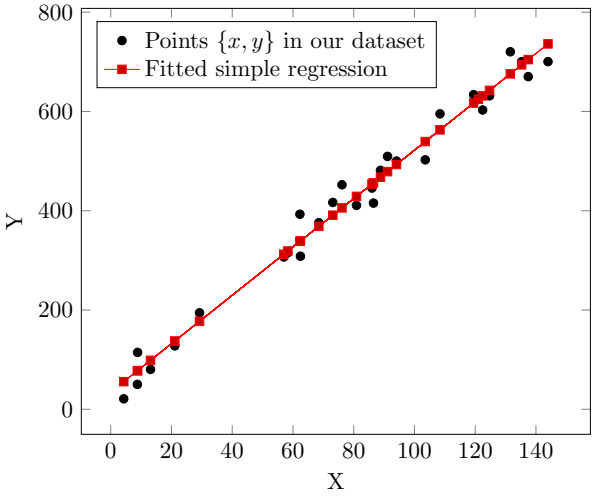

Figure 25: edge dataset 9

Now, lets run the simple regression algorithm for every edge on the graph. After running the algorithm, we obtain the a non-static matrix where every generator function is a linear function of the form $ax + b$. Since it can not bee seen which edge is which, this is a list of the different linear functions that the machine learning algorithm predicted.. There are 10 datasets since it is a 5 vertices complete graph.

- Dataset 1: $-3.39002x + 20.93887$
- Dataset 2: $1.80194x + 55.57595$
- Dataset 3: $2.04345x + 25.80559$
- Dataset 4: $1.73739x + 62.46386$
- Dataset 5: $-5.49907x + 80.34571$
- Dataset 6: $3.44740x + 63.95510$
- Dataset 7: $-1.16348x + 26.18886$
- Dataset 8: $4.05629x + 51.93001$
- Dataset 9: $4.70058x + 28.86789$
- Dataset 10: $-4.28734x + 50.57979$

[26] The grahps are plotted after taking care of outlayers

Since all functions are linear using a modification of the traditional epsilon error function is what we will do. However, let this be the case, it will leave quite a boring graph because when an edge goes "out" of the image, it will not get back to it. This will leave us with a graph that is not really interesting. To make it a little more "exceptional", depending on the situation, we can consider a boundary and if the function is in that boundary, it will be drawn, otherwise, eliminated. Therefore, the error function is the following:

$$Error(e_k, x) = \begin{cases} 1 \text{ if } |f_k(x) - 40| \leq 35 \\ 0 \text{ if } |f_k(x) - 40| < 35 \end{cases} \tag{16}$$

Now that the non-static graph is defined for this concrete dataset, we can generate the 3D image to see it.

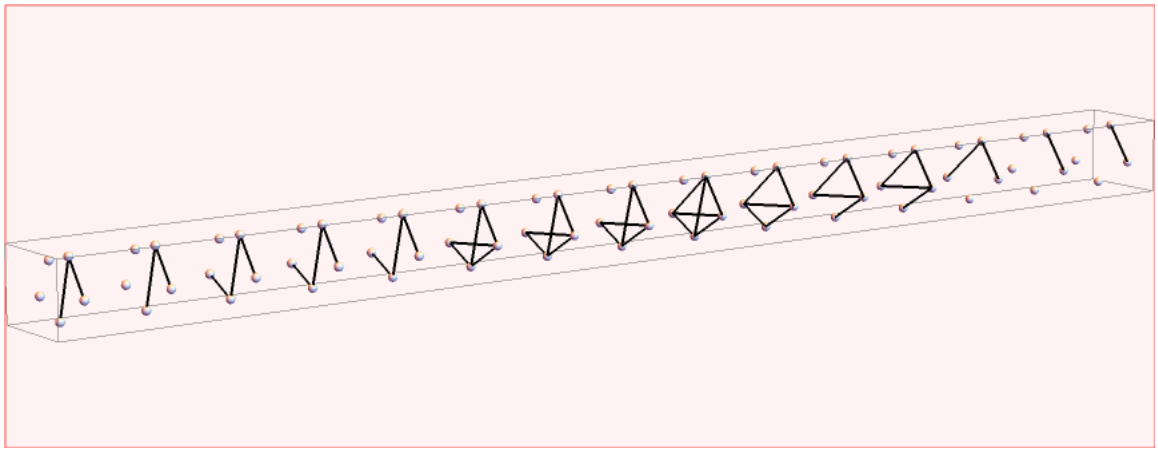

Figure 26: visualization of the graph

4.2.4 Pearson correlation to check the correctness of the generator function

During all these past sections, machine learning and statistics have been used to find a linear tendency of a dataset. If the number of edges is small and one dimensional, it is possible to check by hand all the edge datasets to see whether they follow the requirements of linearity. However, when the number of independent variables is bigger then 3 or the number of edge datasets is considerably big, this solution becomes nothing but useless.

Having said so, a way to find weather a dataset follows a linear tendency has to be though of.

Fortunately, this method already exists and is called Pearson Correlation coefficient. The proof and complete explanation are advanced and a bit out of topic, because of this, let assume that this coefficient has already been proven to work. The Pearson correlation coefficient r_{xy} is a number between 0 and 1 which indicated the linear correlation that a dataset $\{(x_1, y_1), (x_2, y_2), \dots (x_n, y_n)\}$ has. If this coefficient is closer to 1 the linear dependency is considerable and as is gets closer to 0, it means that there is no linear relation. Let $\overline{y} = \frac{1}{n}\sum_{i=1}^{n} y_i$ and $\overline{x} = \frac{1}{n}\sum_{i=1}^{n} x_i$ then, the Pearson correlation coefficient r_{xy} of our dataset is:

$$r_{xy} = \frac{\sum(x_i - \overline{x})(y_i - \overline{y})}{\sqrt{\sum(x_i - \overline{x})^2}\sqrt{\sum(y_i - \overline{y})^2}}$$

Having said so, this formula is going to be used on every edge dataset of an aspiring non-static graph to see whether linear of multiple regression can be used. If $1 - r_{xy} > \epsilon$, the edge dataset will be considered to be non-linear dependent. To finish, since this this is just a formula, the code to compute the coefficient is just computing all the summations through a for loop and use the formula. This function will be called $PearsonCheck(x)$ and will return true if the dataset x has a linear correlation.

4.3 Multi-variable linear tendency

Now that we have defined a way to deal with the most simple edge dataset and create a non-static graph with it, lets suppose that the dependent variable no longer depends on a single one independent variable x but we have a set of m independent variables that will be called $X = \{x_1, x_2 \dots x_m\}$. Our independent variable dataset is therefore an $n \times m$ matrix where n is the number of given points. Just as we have done in the linear regression, we assume that this $m + 1$ dimensional space dataset follows a linear tendency of an m dimensional hyperplane with the equation $\alpha_1 x_1 + \alpha_2 x_2 + \ldots + \alpha_1 x_m + \beta + \epsilon = y$ where epsilon is again an arbitrary small noise or experimental error.

As good example of a place where this algorithm can be used is in the economics field. Let e_{ij} be a directed edge whose weight represents the benefits of a business interaction between enterprises i and j. In this example a graph could be built between all the enterprises in a niche sector. In this case we would have a dataset of all the interactions between these two enterprises. However,

the benefits depend of various factors such as size of the transaction, country, items involved... Since it is not possible to plot in more than 3 dimensions, to illustrate the multiple regression we will do the case where $m = 2$ so $z = \alpha_1 x + \alpha_2 y + \beta$.

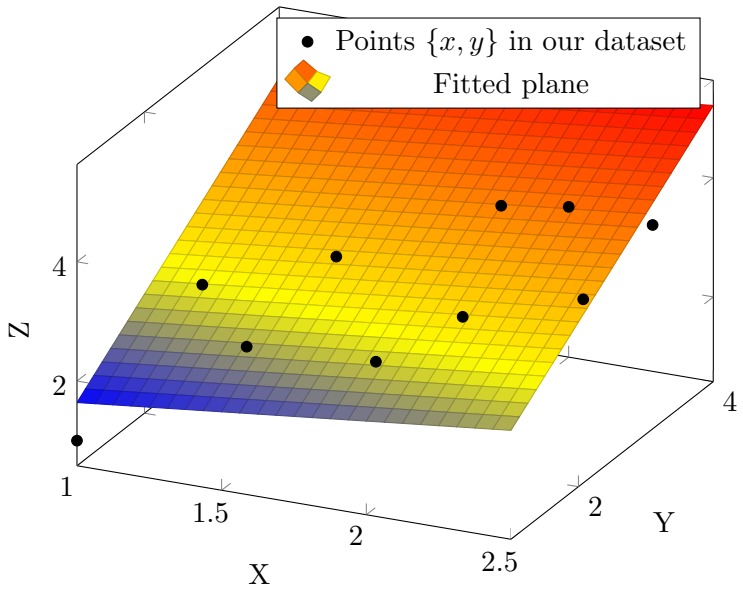

Figure 27: Plot of a linear regression

$$\begin{bmatrix} X & Y & Z \\ 1 & 1 & 1 \\ 2 & 3 & 4 \\ 2.5 & 3.1 & 4 \\ 1.5 & 2.7 & 3 \\ 1.4 & 1.8 & 2.2 \\ 2 & 4 & 3.1 \\ 1.2 & 2 & 2.9 \\ 1.8 & 2 & 2.1 \\ 2.1 & 2 & 3.1 \\ 2.4 & 2.5 & 3.2 \end{bmatrix} \tag{17}$$

Figure 28: Dataset

Proceeding just like in the simple linear regression, to find the generator function, the hyperplane function has to be found. To do so, we will proceed similarly to the simple linear regression and will use the minimization of squared errors.

4.3.1 An expansion on error squared for m dimensions

In this section I will explain my approach to the generalization of the simple linear regression to work with m variables[27]. My approach may not be the most "mathematically" efficient one, but it reduces the problem to something that is computationally solvable with a smart approach, which is good enough for our purposes.

First of all we have to define the dataset with m variables, in the simple regression, y only depended of one x, but this one depends on m x, so we can not longer store all the x values

[27]This generalisation is the way I found to compute the multiple regression

in one dimensional vector. That is why we will store the sequence of variables for a given y_i, $x_{i1}, x_{i2}, \ldots, x_{im}$ in an $n \times m$ matrix:

$$DatasetX = \begin{bmatrix} x_{11} & x_{12} & \ldots & x_{1m} \\ x_{21} & x_{22} & \ldots & x_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ x_{n1} & x_{n2} & \ldots & x_{nm} \end{bmatrix} \quad DatasetY = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix} \tag{18}$$

Now that we have defined how we will refer to the values, we can proceed to generalise the algorithm. Since $\widehat{y}$ is assumed to be linearly dependant on all m variables, we have to express $\widehat{y}$ in a linear function of all of them:

$$\widehat{y} = \alpha_1 x_1 + \alpha_2 x_2 + \ldots + \alpha_m x_m + \beta$$

Having said that we can rewrite the e_i^2 as:

$$e_i^2 = (y_i - \alpha_1 x_{i1} - \alpha_2 x_{i2} - \ldots - \alpha_m x_{im} - \beta)^2$$

which, since we want to minimize the sum of all errors, gives us the following function to minimize:

$$f(\beta, \alpha_1, \alpha_2 \ldots \alpha_m) = \sum e_i^2 = \sum (y_i - \alpha_1 x_{i1} - \alpha_2 x_{i2} - \ldots - \alpha_m x_{im} - \beta)^2$$

To minimise this expression, we have to take the partial derivatives, but since there are $m+1$ variables the entire system can not be written down. To solve the problem, since at the naked eye it can be seen that taking the derivative with respect to α_1, or α_i will be similar, we can take instead the derivative with respect to α_k and β. The first one will give us some sort of parametric function that will return the derivative with respect of α_k for any given k, $1 \leq k \leq m$.

$$\begin{aligned} \frac{\partial f}{\partial \alpha_k} &= 2 \sum (y_i - \alpha_1 x_{i1} - \alpha_2 x_{i2} - \ldots - \alpha_m x_{im} - \beta)(-x_{ik}) \\ &= -2 \sum y_i x_{ik} + 2\alpha_1 \sum x_{i1} x_{ik} + 2\alpha_2 \sum x_{i2} x_{ik} + \ldots + 2\alpha_k \sum x_{ik}^2 + \ldots 2\alpha_m \sum x_{im} x_{ik} + 2 \sum x_{ik} \beta \\ \frac{\partial f}{\partial \beta} &= 2 \sum (y_i - \alpha_1 x_{i1} - \alpha_2 x_{i2} - \ldots - \alpha_m x_{im} - \beta)(-1) \\ &= -2 \sum y_i + \alpha_1 \sum x_{i1} + \alpha_2 \sum x_{i2} + \ldots + \alpha_m \sum x_{im} + 2n\beta \end{aligned} \tag{19}$$

To find the critical points, we set all functions to 0, divide both sides by 2 and express the equation system of $m+1$ variables in matrix notation[28]:

$$\begin{bmatrix} \sum x_{i1}x_{i1} & \sum x_{i2}x_{i1} & \dots & \sum x_{im}x_{i1} & \sum x_{i1} \\ \sum x_{i1}x_{i2} & \sum x_{i2}x_{i2} & \dots & \sum x_{im}x_{i2} & \sum x_{i2} \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ \sum x_{i1}x_{im} & \sum x_{i2}x_{im} & \dots & \sum x_{im}x_{im} & \sum x_{im} \\ \sum x_{i1} & \sum x_{i2} & \dots & \sum x_{im} & n \end{bmatrix} \begin{bmatrix} \alpha_1 \\ \alpha_2 \\ \vdots \\ \alpha_m \\ \beta \end{bmatrix} = \begin{bmatrix} \sum y_i x_{i1} \\ \sum y_i x_{i2} \\ \vdots \\ \sum y_i x_{im} \\ \sum y_i \end{bmatrix} \tag{20}$$

Now we have a really repetitive system of equations to solve, which, to make it a little bit easier for computational purposes we express it as:

$$A \cdot R = \cdot B \tag{21}$$

After the process we have ended up with a complete linear system of equations which is not hard to solve. To so, we will use the gauss-Jordan method.

My first approach to solving this problem was using the traditional way of solving a matrix system of equations which is calculating the inverse matrix and multiplying both sides by it. To compute the inverse matrix I though an interesting recursive algorithm using bit-masks and dynamic programming but it run in $O(n^2 2^n)$ which was too slow, however was more precise than Gauss-Jordan elimination.[29] Gauss-Jordan is not really precise for large datasets because it used a lot of division and because in computers $floats$ can store a limited number of decimals, it loses precision with every division. Using *longdoubles* can improve the performance but it still leaves some considerable errors with large datasets. However, since calculating a matrix inverse with the usual formula $A^{-1} = \frac{1}{|A|} adj(A)^T$ only used one division, my initial method is much more precise. This precision is won at the cost of a very significant computational efficiency because this method can take up to $m = 30$ in 2 seconds and the Gauss Jordan elimination can solve systems up to $m = 300$ in 2 seconds.

[28] We will refer to the matrices with the same name as before, $A \cdot R = B$

[29] See annex for this implementation

4.3.2 Gauss-Jordan elimination

The Gauss method appears, albeit without a proof, in Chinese papers dating back to 179 A.C. In Europe, this method recalls to Newtons notes, from which Gauss got heavily inspired to write the so called Gauss method. Years later, Jordan wrote a variation of the Gauss method which is the one that we will use to solve our system of equations. The algorithm has one major flaw, the accuracy can be compromised by $float$ range because it uses a lot of divisions. This will lead to significant innacuracies for matrices bigger than 40×40. Nevertheless, our system of equations will rarely be so big. This come with a major advantage toward the naive approach, it runs in $O(n^3)$ for square matrices and $O((n+m)n)$ for incomplete systems of equations.

Procedure :

The algorithm works for solving any system of equation of the following form:

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \ldots + a_{1m} &= b_1 \\ a_{21}x_1 + a_{22}x_2 + \ldots + a_{2m} &= b_2 \\ &\vdots \\ a_{n1}x_1 + a_{n2}x_2 + \ldots + a_{nm} &= b_n \end{aligned} \tag{22}$$

This system can be expressed in matrix form as:

$$Ax = b$$

Where A is an $n \times m$ coefficient matrix and x, b are one dimensional vector of the variables and the independent therms.

The algorithm is a sequential elimination of the variables in each equation until the matrix A, in our case of a square matrix, gets reduced to the identity. In this state, the system will have a clear solution. $x_i = b_i$.

The algorithm works in steps. In the first step, we take the first row of the matrix a_1 divide it all by a_{11} and add it to the other rows such that the element a_{1i} is zero.

$$a_i = a_i - \frac{a_{i1}}{a_{11}} a_1$$

At the end of this iteration all the elements of the first column will be 0 except a_11 which is 1. We repeat this process for all column until, in the case of a square matrix it will be transformed the identity matrix. During the iteration it is possible that $a_{ii} = 0 + \epsilon$ which will break the step because it is not possible to divide by 0, therefore, we will just swap this vector with another one that has not been used yet.

NOTE: This algorithm has been inspired by the one shown on CP-algorithm, a famous competitive programming website.

Algorithm 10 Gauss Jordan Elimination

Input: ans, a, n, m

```
where ← {}
for col < m and row < n do
    sel ← row
    for i in range(row, n) do
        if |a[i][col]| > |a[sel][col] then
            sel ← i
        end if
    end for
    if |a[sel][col] < ϵ then
        continue
    end if
    for i in range(col,m + 1) do
        swap(a[sel][i], a[row][i])
    end for
    where[col] ← row
    for i in range(0,n) do
        if i ≠ row then
            c ← a[i][col] ÷ a[row][col]
            for j in range(col,m+1) do
                a[i][j] ← a[i][j] − a[row][j] · c
            end for
        end if
    end for
    row ← row + 1
end for
for i in range(0,m) do
    if where[i] ≠ −1 then
        ans[i] = a[where[i]][m] ÷ a[where[i]][i]
    end if
end for
```

4.3.3 The complete algorithm

To be able to complete the algorithm, assuming that the functions mentioned above have already been implemented, all the sums that correspond of the coefficients have the be calculated. Finally all the algorithms have to be put together, thing that will give us a vector of size $m+1$ with all the respective coefficients.

First of all, the dataset has the be imported, which is easily done reading it from a *.txt* file with a loop. After we have to calculate all the sums. To do so, as they are very repetitive we implement a function $calculate(k)$ that returns a vector containing all the sums of the k^{th} equation.

$$calculate(k) \rightarrow \begin{bmatrix} \sum x_{i1}x_{ik} & \sum x_{i2}x_{ik} & \dots & \sum x_{im}x_{ik} \end{bmatrix} \tag{23}$$

Therefore, the needed matrix is the following:

$$A = \begin{bmatrix} calculate(1) \\ calculate(2) \\ \vdots \\ calculate(m) \\ \sum x_{i2} \quad \sum x_{i2} \quad \dots \quad \sum x_{im} \end{bmatrix} \tag{24}$$

Now that the system matrix is calculated, it has to be joined with the independent therm vector B and inserted into the Gauss function which will return us the answer vector.

$$B = \begin{bmatrix} \sum y_i x_{i1} \\ \sum y_i x_{i2} \\ \vdots \\ \sum y_i x_{im} \\ \sum y_i \end{bmatrix} \tag{25}$$

$$ans = gauss(A, B) \tag{26}$$

This will yield us an answer vector ans where ans_i is α_i for all $1 \leq i \leq m$ and ans_{m+1} is β.

Algorithm 11 Complete multiple regression

Input: calcDet,calcAdjoin

dataset$\leftarrow$ $GetInput$ ▷ $O(nm)$

$A \leftarrow calculate(dataset)$ ▷ $O(m^2n)$

$B = calculateB(dataset)$ ▷ $O(nm)$

$V = A \cup B$ ▷ $O(1)$

$ans \leftarrow \{\}$ ▷ $O(1)$

$gauss(V, ans)$ ▷ $O(m^3)$

Output: $ans = [\alpha_1, \alpha_2 \ldots, \alpha_m, \beta]$ ▷ Final time complexity: $O(m^2n)$, Memory complexity $O(mn)$

4.3.4 Resulting graph

Let $MultipleRegression()$ be the algorithm just implemented above which takes as parameters an edge dataset and returns an array $ans = [\alpha_1, \alpha_2 \ldots \alpha_m, \beta]$ with the coefficients corresponding to the regression hyperplane. For the specific case where $m = 1$, it is the linear regression and all the coefficients of the generator function can be assigned manually, however, since it is possible to have multiple variables, a program has to be written to execute the prediction. To do so the final array ans has to be iterated to find the predicted cost and calculated accordingly to the formula below.

$$f_{ij}(x_1, x_2 \ldots x_m) = ans_1x_1 + ans_2x_2 \ldots ans_mx_m + ans_{m+1} = ans_{m+1} + \sum_{i=1}^{m} ans_ix_i$$

Consider the edge dataset $data_{ij}$, then, the final generator function will be the following:

Algorithm 12 Multiple variable generator function

Input: MultipleRegression(),dataset

functionGeneratorFunction($x_1, x_2 \dots x_m$)

ans $\leftarrow$ MultipleRegression(dataset)

cost$\leftarrow$ 0

for i **in** range(1,m) **do**

cost $\leftarrow cost + ans[i]x_i$

end for

return cost

end function

In this case, since the functions are linear, it will behave similarly to a Simple Regression graph, however, it admits a variety of different dataset and is the basis of the most important edge prediction algorithm that will be used. Unfortunately since plotting the graph takes $m + 1$ dimensions, it is not possible to plot any example of a multiple regression graph, however, the algorithm is going to get tested on some datasets for the $m = 2$ case but it will not be possible to see the final graph.

4.3.5 Multiple correlation coefficient

Just like in the simple regression, with big graphs, computing whether an edge dataset fits in the description of multi-variable linear tendency can be hard to do by hand. Fortunately, there exists an expansion on the Pearson's correlation coefficient called Multiple correlation that enables to compute this coefficient R squared for multi-variable datasets. Just like before, albeit without a proof, this coefficient will be used with the purpose of detecting the valid edge datasets. Consider the dataset of n independent variables x_i. Let r_{ab} be the Pearson coefficient between the variables a and b, $c = (r_{x_1y}, r_{x_2,y} \dots r_{ny})$ and A be an $n \times n$ matrix such that:

$$A_{ij} = r_{x_i x_j}$$

Then, the squared of the multiple correlation coefficient is defined by:

$$\mathbf{R}^2 = \mathbf{c}^\intercal A^{-1} \mathbf{c}$$

By looking at the formula, it can be observed that if the prediction variables x_i are not correlated in any way, A is just an approximation of the identity matrix. Because $r_{xx} = 1$ and $r_{xy} \approx 0$ if they are not correlated. This means that if it can be assumed that there is no correlation between independent variables, the formula can be reduced to:

$$\mathbf{R}^2 = \mathbf{c}^\intercal \mathbf{c}$$

This special case makes it easy to compute because matrix multiplication for vectors is computed with a for loop in $O(n)$ which gives us a really fast program. However, let this not be the case, a matrix inverse has to be computed. Even though it is no easy task I decided to not explain my approach in the body of the monograph and the function $Correlation(A)$ that returns the multiple correlation coefficient will be assumed to be implemented for the following sections. The algorithm that I used is not hard, it uses bit masks and dynamic programming to avoid repetition on the recursive definition of determinant. And finally used the well known formula for the inverse of a matrix to compute it $A^{-1} = \frac{1}{|A|} adj(A)^T$.

4.4 A new approach to the Multiple Regression

After dealing with the linear edge datasets, there is a dataset that does not follow a linear tendency. To manage it, assume that on the interval it is needed, it follows a polynomial of degree k. In this moment, it is important to point out that any dataset of size n can be fitted in a unique polynomial of degree $n-1$ without any error. However, a perfect fit would not be accurate because we just want to interpret the tendency of the dataset and not the exact patron, see figure 20.

Assuming it is a polynomial of degree k, the predicted y can be expressed as following $\widehat{y} = \alpha_1 x + \alpha_2 x^2 \dots \alpha_k x^k + \beta$. We proceed with the minimization of error squared. Therefore:

$$\sum e_i^2 = \sum (y_i - \alpha_1 x + \alpha_2 x^2 \dots \alpha_k x^k + \beta)^2$$

It is still not known which k fits best to the dataset. However, it can be seen that a similar pattern than the multiple regression is followed, but instead of taking different values of j for x_{ij} take the same x_i is taken raised to the j^{th} power. Having said so, everything points out at the

possibility of reusing the multiple regression algorithm.

To do so, we have to notice that the variables: $\alpha_1, \alpha_2 \dots \alpha_n$ and β are the same. The only difference is that instead of x_{ij} we have an x_i^j. To proceed with the minimization we need and $n \times k$ dataset, which we do not dispose. However, it is possible to transform the $n \times 1$ dataset at our disposal to make it $n \times k$ which will be used to find the function coefficients vector. This transformation of the matrix is going to be called polynomial transformation and will transform our linear dataset to a matrix $A_{n\times k}$ such that $MultipleRegression(A)$ returns the polynomial coefficients of degree k.

$$Dataset = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} \longrightarrow \begin{bmatrix} x_1 & x_1^2 & \dots & x_1^k \\ x_2 & x_2^2 & \dots & x_2^k \\ \vdots & \vdots & \ddots & \vdots \\ x_n & x_n^2 & \dots & x_n^k \end{bmatrix} \tag{27}$$

After transforming the dataset it can be processed in exactly the same way as a multiple regression to get the final values of $\alpha_1, \alpha_2 \dots \alpha_n, \beta$ which yield us the generator function for these kind of edges.

4.4.1 Prove of the validity of this approach

Now, before proceeding, it has to be proven that following this process actually minimizes the k degree polynomial sum of error squared. Let M and V be the dataset of a multiple regression and the transformed dataset by the process mentioned above; then, when fitting a polynomial function we can express the predicted y as:

$$\widehat{y} = \alpha_1 x^1 + \alpha_2 x^2 \dots \alpha_k x^k + \beta$$

Then, the function to minimize is:

$$\sum (y_i - \alpha_1 x_i^1 + \alpha_2 x_i^2 \dots \alpha_k x_i^k + \beta)^2$$

By definition, minimizing the sum of errors of V, since $M_{ij} = x_{ij}$ means that $V_{ij} = x_i^j$, x_{ij} is equivalent to x_i^j, therefore, we can substitute them in $\sum e_i^2$ of a multiple regression:

$$\begin{aligned} \sum e_i^2 &= \sum (y - \alpha_1 x_{i1} - \alpha_2 x_{i2} \dots - \alpha_m x_{im} - \beta)^2 \\ &\to \sum (y - \alpha_1 x_i^1 - \alpha_2 x_i^2 \dots \alpha_m x_i^m - \beta)^2 \end{aligned} \tag{28}$$

This substitution gives us the sum of errors squared of a polynomial regression with $k = m$, meaning that what we are really doing is minimizing the correct function which proves that our approach is in fact correct.

Developing this expression, just like in a multiple regression, we obtain the following matrix system to solve, which will be solved by the exact same algorithm.

$$\begin{bmatrix} \sum x_i^2 & \sum x_i^2 x_i & \dots & \sum x_i^k x_i \\ \sum x_i x_i^2 & \sum x_i^4 & \dots & \sum x_i^k x_i^2 \\ \vdots & \vdots & \ddots & \vdots \\ \sum x_i x_i^k & \sum x_i x_i^k & \dots & \sum x_i^{2k} \\ \sum x_i & \sum x_i^2 & \dots & \sum x_i^k \end{bmatrix} \begin{bmatrix} \alpha_1 \\ \alpha_2 \\ \vdots \\ \alpha_k \\ \beta \end{bmatrix} = \begin{bmatrix} \sum y_i x_i \\ \sum y_i x_i^2 \\ \vdots \\ \sum y_i x_i^k \\ \sum y_i \end{bmatrix} \tag{29}$$

All in all, we come to the conclusion that scaling a dataset to the i^{th} power and interpreting just like a multiple regression dataset fits the i^{th} power polynomial.

4.4.2 Polynomially transforming a vector

Formally speaking, polynomially transforming a vector $V = [x_1, x_2 \dots, x_m]$ to the k^{th} power means transforming it to an $m \times k$ matrix M such that $M_{ij} = x_i^j$. To generate the scaled dataset efficiently we need one intuitive observation. It is well known that $a^k \cdot a = a^{k+1}$, therefore:

$$M_{ij} = \begin{cases} M_{ij-1} \cdot V_i \text{ if } j \neq 1 \\ V_i \text{ if } j = 1 \end{cases} \tag{30}$$

With this recursive definition, for each row in the dataset we can iterate through all k columns and calculate its corresponding value based on the previous value (which will for sure have been calculated). Assuming multiplication is done in $O(1)$, this gives us a time complexity of $O(nk)$.

Algorithm 13 Polynomially scaling a vector

Input: dataset,n ▷ n is the size of the dataset

```
function ptransformdataset,k
    newdataset ← [n][k]
    for i in 1 to n do
        for j in 1 to k do
            if j = 1 then                                   ▷ second case
                newdataset[i][j] = dataset[i]
            else                                            ▷ first case
                newdataset[i][j] ← newdataset[i][j − 1] · dataset[i]
            end if
        end for
    end for
    return newdataset
end function
```

4.4.3 Computation of the polynomial degree

The big issue with this approach is computing the optimal degree to use to scale the dataset. This can be done in a naive way by iteration through 30 degrees and manually checking whether they correspond to the edge dataset. However, this can be done computationally using the multiple correlation.

We iterate from 1 to a chosen limit L. In each iteration, to know if a given degree k of the polynomial is correct, we need to check whether the scaled matrix follows a k dimensional linear tendency. As k gets larger, the function will fit better and it will continuously get more linear, therefore, the multiple correlation coefficient will get increasingly bigger. To know when to stop increasing k, if a scaled matrix to the k^{th} degree has a correlation coefficient greater than 0.7 if will be considered to have a significant correlation and the algorithm can be stopped. This leaves us with this final algorithm:

Algorithm 14 Complete Polynomial fitting algorithm

Input: calcDet,calcAdjoin, Pscaling

dataset← $GetInput$ ▷ $O(n)$

for k **in** 1 **to** L **do**

polyDataset ← $Pscaling(dataset, k)$

$A \leftarrow calculate(polyDataset)$ ▷ $O(k^2 n)$

$B \leftarrow calculateB(polyDataset)$ ▷ $O(nk)$

$correlation \leftarrow Correlation(A \cup B)$ ▷ Calculate correlation

if $correlation > 0.75$ **then** ▷ evaluate correlation

return$MultipleRegression(A \cup B)$ ▷ $O(m^3)$

end if

end for

Output: $R = [\alpha_1, \alpha_2 \ldots, \alpha_m, \beta]$

4.4.4 The predicted edge function

At the end, will be have a new way to define edge functions which is much more efficient and applicable to real life than the other two but, in the other hand, it requires significantly more computational time and can lead to really easy over-fitting. At the end, if ans is the array of coefficients, an edge function $f_{ij}(x)$ will therefore be defined as:

$$\begin{aligned} ans &= MultipleRegression(Ptransform(dataset_{ij})) \\ f_{ij}(x) &= ans_{m+1} + \sum_{i=1}^{m} ans_i x^i \end{aligned} \tag{31}$$

This will yield us a graph that behaves like the datasets it represents, and more importantly, it partially solves the major problem of these graphs, finding the corresponding edge function for all edges. In this moment it is important to note that exist much more complex and versatile machine learning algorithms that would work to predict the edges of a graph, which makes the possibilities of improvement and utilities of non-static graphs limitless.

5 Modular implementation of a non-static graph

Now that we have defined everything we need of non-static graphs, to use it in our specific situation we have to first program it. For the implementation of a non-static graph, we will use Object orientated programming[30], also known as modular programming. I chose to use Classes because even though it is perfectly possible to do it with imperative or functional programming (and will probably result in a shorter code), I think that using classes will result in a much more organized code which is better to understand, faster to read, easier to debug and more reusable. Another reason why I used classes is because this project is an introduction to this new concept of graph, therefore, I aim to build the basis of the theory. The classes built here are designed in such way that they can be *extended*[31] into subclasses by any user to improve non-static graphs and add functionalities. The general code will be structured as following; a graph class that will contain all the information about the graph and methods and an edge class which will basically contain the generator function and the machine learning algorithms.

5.1 Edge Class

The Edge Class has to have two main methods, given a value of t return the corresponding edge cost and given an edge dataset execute the machine learning algorithms to fit the generator function. It has also another method which allows to choose an arbitrary generator function for the purpose of testing.

[30] It is usually referred as OOP and is a way to structure the code using Classes.

[31] Extension is an OOP technique that allows to create subclasses that have the properties of the class they are derived but can have some additional functionalities

Algorithm 15 Edge Class

Class EDGE

declaration

Coefficients $\leftarrow [0, 0 \ldots, 0]$

end declaration

methods

void **Constructor**(*type*)

float **generator function**(x)

void **find**(*dataset*)

end methods

end Class

The fitting function, $find(dataset)$ has the purpose of finding the generator function given a dataset. Assuming the needed algorithms are already implemented (simple linear regression, multiple regression and Polynomial fitting), it is only a matter of checking the cases mentioned in 4.1 Required dataset and execute the corresponding algorithm. As it has been said, we check linearity with the Pearson correlation coefficient, if it gives a positive evaluation, the multiple linear regression algorithm is executed, otherwise, we execute the polynomial fitting algorithm.

Algorithm 16 fitting function

function find(dataset, case)

if $PearsonCheck(dataset)$ ==**true then**

$coefficients \leftarrow MultipleRegression(Dataset)$

else

type $\leftarrow$ *poly*

$coefficients = PolynomialFitting(dataset)$

end if

end function

The generator function is also a really basic function. It checks the type of the function *linear* or *polynomial*, then finds the value of the edge with the previously computed coefficients.

Algorithm 17 generator function

```
function generator function (X)
    declare cost
    if correlationCheck(dataset) then
        cost ← β
        for α_i in alphas do
            cost ← cost + α_i X_i
        end for
    else
        for α_i in alphas do
            cost ← cost + α_i X_0^i          ▷ there is only one x if polynomial.
        end for
    end if
    return cost
end function
```

5.2 Graph Class

The graph Class is the important part of the code, an instance of this class should have a complete functionality of a non-static graph. Inside the class it should store the $n \times n$ adjacency matrix[32] of the graph $adjMatrix$ and the size of the graph n. As methods it holds the *costedge* which returns the cost of the given edges at a given x, and its error functions. Note that a single non-static graphs can have various error functions. Our graph will also have a random function which generates a totally random graph with random generator functions and edges.

It is important to say than when an instance of the graph is created, it is not fully trained, meaning that the generator functions are defined as *null*, this means that the user, when implementing it has to go through all the edges of the adjacency matrix and call the fit function. All of this is done inside the constructor method.

[32]Remember that the adjacency matrix holds the generator functions, this means that each element of the matrix the an instance of the Class EdgeFunction

Algorithm 18 Non-static graph Class

Class NoneStaticGraph

declaration

adjMatrix $\leftarrow [[EDGE_1], [EDGE_2] \dots [EDGE_{\binom{n}{2}}]]$

end declaration

methods

void **Constructor**(*size*, *datasets*)

float **edgeCost**(i, j, x)

boolean **basicError**(i, j, x)

boolean **MSTerror**(i, j, x)

void **randomGeneration**()

end methods

end Class

The edge cost is just a bridge to the generator function. It will call the generator function of the element $adjMatrix[i][j](x)$ and return the value. The basicError function is an error function that considers a boundary ϵ and returns true if $f(x) < \epsilon$ and false otherwise. The MST is the error function to build a tree graph which will use the algorithm mentioned previously.

Algorithm 19 basicError function

function basicError(i,j,x)

if $adjMatrix[i][j](x) \geq \epsilon$ **then**

return $false$

else

return $true$

end if

end function

Algorithm 20 MST error function

```
function MSTerror(i,j,x
    par ← [1, 2, 3 . . . n]                    ▷ Creates the required for the MST algorithm
    edges ← []
    for a in range(1,n) do
        for b in range(1,n) do
            edges ← edges ∪ adjMatrix[a][b](x)
        end for
    end for
    MST ← MST(edges)                           ▷ executes the MST algorithm
    if e_ij ∈ MST then                         ▷ Checks if the given edges is in the MST.
        return true
    else
        return false
    end if
end function
```

It is important to say that this pseudo code is just an overview and are implemented in a way that results visual and short. In the C++ code, some of the functions are a little more optimized, for example, the MST algorithm is only called once and not for every edge. In the real implementation, the current MST is stored and only recalculated if the x value has changed. To end this short section, I will run through an example on how to set and prepare a non-static graph using both classes mentioned above.

"Without data you are blind and deaf and in the middle of the freeway"

Geoffrey Moore

This page has been intentionally left blank

6 Connecting underdeveloped civilizations in an efficient way

As it is well known, some small towns in Africa have not had the resources to invest in road building to communicate with other towns and are therefore very little developed. This makes them enter an infinite loophole where, because of not having communication they can not enhance their economies and in a world dominated by capitalism they are unable to develop. This makes then unable to build roads and therefore entering in the vicious circle of poverty. Moreover, this lack of communication is also one of the reasons that most of people in these countries are unable to educate themselves properly because they are unable to get to developed cities or any help from the outside.

To help this communities, what is often done is giving them resources to enable their survivability. However, if they are only helped at this basic level, they will never be able to develop. After observing the reasons of success of many first world European countries I realised that development does not come from strong politicians nor successful scientists, but from communication between neighbour underdeveloped cities. The point that I want to make is that a town that is isolated from any other will not create a modern economic system and will neither be able to make advancements in technology because all of this is done in community. France would not be a first world country if they had been isolated from the rest of Europe.

Having said so, what I want to do in this first part of the project is use a graph algorithm to connect all the cities in a Region of Africa in an optimal way so they are able to have clear paths between each other and start a modern economy that will, in the long therm help them get out of the loophole.

6.0.1 Overview of the problem

The problem is the following, imagine we have an underdeveloped region which has a number of isolated towns that do not dispose of barely any connection to move around. Consider now an aerial image of the entire region and consider the Cartesian plane of coordinates where the origin, $(0, 0)$ is at the bottom left corner of the photo. Now, lets denote every town T_i with

some coordinates (x_i, y_i). The objective is to connect all the cities with some roads such that the building cost, which is directly proportional to the sum of distances of each road is minimal. In this project, we will try to be realistic in order to plan a project that could be put in place by a first world country. In this case we see that for the inhabitants of the chosen region, the optimal way to connect all cities would be to build a road from each city to any other city. However, this massive project would cost billions and billions of dollars which will throw investors back. In its place, I propose an alternative problem, find the cheapest road structure of a set of cities such that it is possible to go from any city A to any other city B with the built roads. The problem is that this heuristic connection could lead to some big distances between two arbitrary cities T_a and T_b, but overall, it is a pretty interesting building project that sells good to investors.

6.0.2 Building the graph

Now that the problem is well know, we have to solve it. First of all, we have to interpret every city on the map as a vertex of the graph so let v_i be the vertex corresponding to the i^{th} city. Lets consider now the complete graph of this set of vertices where the cost of each edge is the cost in Euros that would take to build this road. As an example, I have taken 6 arbitrarily chosen cities of Botswana and drawn the corresponding complete graph.

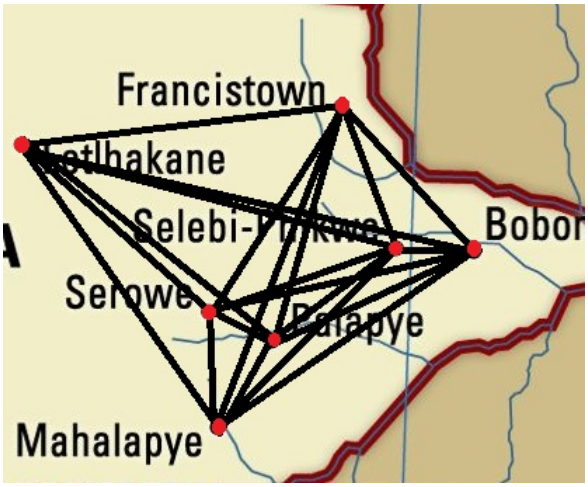

Figure 29: Example graph of 6 arbitrarily chosen cities of Botswana

Consider we have n cities and the complete graph created is $K_n = (V_n, E_n)$, then, the problem reduces to finding $G = (V, E)$ such that $V = V_n$ and $E \subset E_n$ and the sum of weights of the edges

of G is minimum out of all possible E.

To get the cost of each edge, we approximate the building cost for meter to be constant through the country- Then, we know that the cost of road for kilometer is directly proportional to the length of it. Let C be the cost of a road and l the length, we can therefore express C with the following expression (where k is the cost per meter):

$$C = k \cdot l$$

Keeping this in mind, we want to minimize:

$$\sum_{i=1}^{n} C_i = \sum_{i=1}^{n} k \cdot l_i = k \sum_{i=1}^{n} l_i$$

Since k is a constant, the problem reduces to minimize the sum of lengths of all edges of the the subgraph. As a second observation, it can be seen that the resulting subgraph will always be a tree. This is easily proved by contradiction. Let G be the subgraph of our complete graph such that our condition is minimal. Suppose now that G is not a tree. By the definition of tree, we know that it must contain a cycle, therefore, one edge of G can be eliminated without disconnecting any vertex. Hence, G is not optimal and we have a contradiction.
Now that this two observations have been made, the problem has reduced to finding the subtree of the complete graph of cities such that the sum of all edges cost is minimum. Taking a glance at the error function that we created to make non-static trees, we see that it is exactly the same problem and we already have a solution. To find this subgraph, we have to construct the minimum spanning tree of the graph. However, the algorithm that we used required to fill an array with all the edges of the graph, which in our case it is $\binom{n}{2} \approx n^2$. Because of this, running Kurskal's algorithm will be inefficient in therms of memory and time complexity, since it runs in $O(mlog(m))$ with $m \approx n^2$ which turns it into $O(n^2log(n))$. This $log(n)$ factor and $O(n^2)$ space complexity would not matter if we wanted to connect 20,1500 or 10^4 towns. However, even if there are not this many towns, it can be of the region's convenience to not only connect towns but also involve important zones to the roads such as water wells, lakes, cultivated zones ... There are a lot of zones like this in big regions in Africa, and the number of nodes will therefore escalate very quickly which will therefore make it impossible to store in an array and will run

approximately 60 times faster than Kurskal. By luck, there is a solution to the problem that runs in $O(n^2)$ and takes $O(n)$ memory complexity, Prim's algorithm.

An efficient connection with Prim's algorithm Since this problem has already been solved, even if this algorithm is significantly different than Kurskal's, I will not go into depth in the explanation. Suppose we are given an array a of length n where $a_i = (x_i, y_i)$, the coordinates of the place T_i. Therefore, the cost to connect a_i with a_j will be $\sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$ (the euclidean distance between the two points on the map). The idea behind it is very simple, fore every not selected vertex store the minimum edge to a selected vertex. It starts by selecting a vertex, with the minimum stored edge, adds it to the MST, recomputes the minimum edge array and repeats the process until $n - 1$ edges have been selected.

Algorithm 21 Prim's algorithm

Input: minEdge[ ], selected[ ], MST[ ]

```
for i in range(0,n) do
    v ← −1
    for  j in range(0, n) do
        if not selected[j] and w(minEdge[j]) ¡ w(minEdge[v]) then
            v ← j
        end if
    end forselected[v] ← true MST ← MST ∪ v ∪ minEdge[v]
    for j in range(0,n) do
        if dist(v,j) ¡ w(minEdge[j]) then
            minEdge[j] = e_vj
        end if
    end for
end for
```

Note: In the pseudo code above, dist(i,j) returns $\sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$. which is the cost of the edge.

At the end, we have ended with an algorithm that runs in $O(n^2)$ time complexity and $O(n)$ space

complexity which is pretty good for our intentions. At the end, to fins this optimal connection between cities we just have to get the data of its coordinates, thing that is not that hard to obtain.

6.0.3 Final results

Now that we have the algorithm, lets apply it to a specific case, as a region, I chose a considerably small region of Ethiopia. Which accounts for 38 important cities and regions. I got an air image from Google Maps, downloaded the photo and imported it into Wolfram Mathematica to get the specific coordinates of the cities. I inserted the coordinates into the algorithm and drown the resulting tree on the map, as can bee seen below.

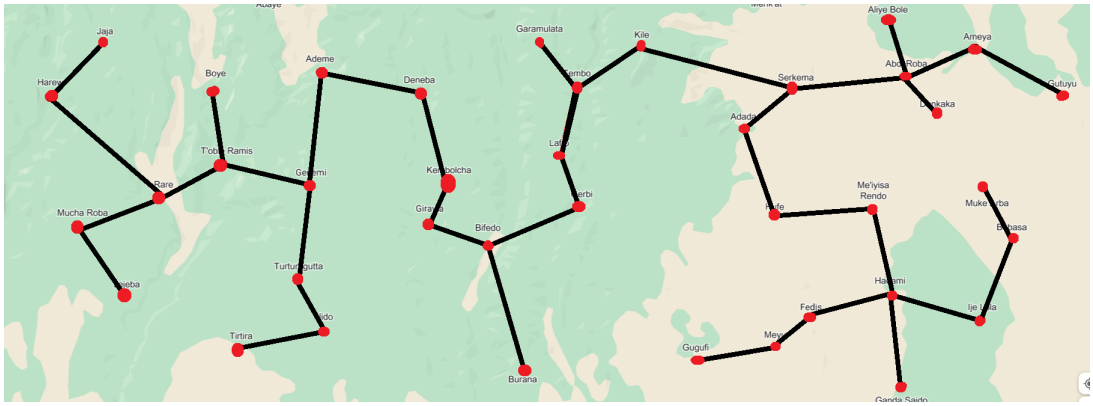

Figure 30: Final roads

Observing the result it can be seen that the connection is not as efficient as we would want it. Regardless, this is the most optimal sub graph that connects all the corresponding cities and at the end, it is what we are selling to big investors.

6.0.4 How to improve on this method

It can clearly be seen that this algorithm does not take into account if cities are important, big or the specific needs of each city. To improve the algorithm, it could be great to let a human observe and analyse the situation and choose to add other connections if necessary. This application would only be a pre-calculation to save time and money of the true professional and

have an objec tive first configuration to avoid biases towards known cities. This is because if all the process was done by a human, it could lead to huge biases to small and unknown cities that would lead to bigger inequalities, thing that goes against our ideals.

On the other hand, to take into account the specific necessities of every city, we could use a non-static graph where the cost function of each edge depends on the $length \cdot k$ where k is a value predicted by one of the machine learning algorithms taking into account all the relations of the past years between those cities. The Machine learning would learn from a dataset of number of transactions, citizens... and predict that value k. However, I do not dispose of any database that allows me do develop further on non-static graph. This is the reason why I have only treated non-static graphs from the theoretical point of view and will only go through the theoretical process on how to use them in the next section.

7 Finding the shortest path with a non-static graph

In this section we are going to discuss how to find the shortest path from one city to another taking into account traffic as a function of time. Imagine there is a person that stands in the point A of the map and desires to travel effectively to another point B. He wants therefore to compute the least time consuming set of roads to take. But what is the shortest path? The shortest path can be viewed from a physical point of view and it can be said that the shortest path from A to B is the sequence of rods such that the sum of all distances i minimum. However, if there can exist a path such that the total distance is higher but the average velocity is also higher which results in a faster travel. We can therefore define the shortest path from A to B, in our problem, to be the sequence of roads such that the time taken to travel through them is minimum. As has been raised in section 2.5, this problem has already a solution, we interpret the situation as a graph, as will be seen in the following section, run Dijkstra's algorithm and it immediately yields the shortest path. However, the objective of this section is to prove the usefulness of non-static graphs by solving the same problem but taking traffic into account, which is the limitation of graph theory raised in section 3.1.

7.1 Interpreting the situation

It can be seen that this problem is very similar to the shortest path problem mentioned in section 2.5 we just have to interpret the map as the correct graph. To do so we will take each road intersection to be a vertex and each road to be the edges. As an example, let this part of Barcelona be the interpreted location, the corresponding graph will be the shown in figure 31

Figure 31: Barcelona Graph

As said before, the weight of each edge would be the time taken to travel from one point to another. Therefore, if the distance of the edge is x and the velocity allows over it is v, them the weight of the edge will be $\frac{x}{v}$.

In this example we have taken a small sample of terrain, let this not be the case, interpreting the graph is a little big harder. Scaling up the dimensions, consider now that it is wanted to travel from one point of Barcelona to the other side of the city. There are thousands of different road intersections which would lead to a massive graph that would slow the algorithm and take too much space to store. In this case, we have to work through an approximation of the graph. We will only interpret as vertices places of interest such as big establishments and important road intersection and we will only consider connections between them as edges. Consider as an example the graph of figure 32 where I have considered points of interest: metro stations, train stations, famous buildings, main and secondary road intersections.

Figure 32: Approximation graph

It can clearly be seen that the resulting graph is just an approximation of the real graph, however since vertices are relatively close to each other, the shortest path approximation will be quite good. Note that the drawn edges do not follow the path of the roads, however, the cost of the edges is calculated using the real road length. These lengths can be computed running a local Dijkstra's algorithm on a graph similar to the first case of that local zone.

To clarify, this specific case, computationally speaking, can be interpreted just like the first example, but it is interpreted as an approximation for the sake of an example.

7.2 The weight of each edge and generator function

It has already been said that the weight of the graph has to be the time taken to travel through the edge. Had we not wanted to take into account traffic, our weight would be $\frac{x}{v}$ where v is the average velocity in that road. However, traffic changes everything because in an instant a it is possible that it would take t_0 time to travel to edge but in another instant b the time could be $t_1 \neq t_0$. Having this situation, the objective is to be able to approximate the time that would take in a concrete time of the day the time that would take to travel through that edge. This leaves us with a difficult problem to solve precisely because traffic is unpredictable with small

datasets. Consider now that we dispose of a statistically representative dataset of the traffic as a function of time from 0 to 24 hours of each day of the week. Then, it is obvious that it will follow a pattern because society is all about going to work the same day at the same time each day of the week. Keeping this in mind, it could be possible to approximate the weight of the edge using machine learning.

7.2.1 The weight for small voyages

For small distances, due to the unpredictability of traffic on the specific case, we will use a traditional graph. A small voyage takes about 2 hours which will nearly never change the traffic, therefore, the generator functions of each edge will be approximately constants which leaves us a traditional graph. At the end, this small case will be a perfect example on how to use Dijkstra's algorithm in the traditional way. The graph of the city is created and the algorithm is run on it to get the final path. Keeping in mind that traffic influences a lot in the time taken to get to a location, this algorithm will only be accurate in the days that there is not a lot of people on the streets, however, on Christmas eve it will not do accurate predictions.

All in all, for small distances, the algorithm will have two steps, create the graph, run Dijkstra's algorithm on it and finally interpret the result to get the shortest path. All of this can be done with the explained in other sections.

To improve this naive approach, it is possible to predict the average cost as a function of the day with the machine learning algorithms learned in anterior sections. However, this would not lead to a non-static graphs because the travel is not more than a few hours long. This means that the graph would not have time to change having such general predictions. At the end, we end up with a traditional graph that is generated with machine learning at the start of the run. If we want to shortest path in Wednesday it will predict the edges for Wednesday and yield a constant graph.

7.2.2 Generator function for long distances

On the other hand, when we are dealing with long distances, traffic can accurately be predicted with the discussed algorithms which will lead to good results if and only if the dataset if statistically representative.

Dataset:

The edge dataset that will be used is a 7 day dataset that describes the velocity taken though that edge road at a specific time of the day. As a general example, the dataset D is a two row matrix such that the independent variable is the time and the dependent variable is the average speed that this car had over the entire road.

$$dataset = \begin{bmatrix} v_1 & v_2 & \dots & v_n \\ t_1 & t_2 & \dots & t_n \end{bmatrix} \tag{32}$$

This will leave us with an edge function with a polynomial tendency which can be approximated using the polynomial fitting algorithm. It has to be said that to pull out good results using this approach an insane amount of data is required for each edge because the velocities of millions of cars that travel through a road in a week have to be recorded. This is because to make it accurate at least 40 or 50 car speeds have to be recorded each hour and there are $164h$ in a week which gives 8400 velocities for each edge. At the end, if the graph is relatively small, 10^4 edges, it will end up needing $8.4 \cdot 10^7$ recorded velocities for graph.

Calculating the real weight of an edge

We have now a dataset with the velocities which thanks to the polynomial fitting algorithm we have managed to approximate the function of velocity through the edge which on the interval of the dataset $t \in [0, 24 \cdot 7]$ (in hours) is:

$$v(t) = \alpha_1 t + \alpha_2 t^2 + \ldots + \alpha_m t^m + \beta$$

Nevertheless, the generator function has to be the function of time not velocity. To find it, it is well known that given the function of the modulo of velocity, Δx between two instances $t = a$ and $t = b$ is given by:

$$\Delta x = \int_a^b v \, dt$$

Keeping this on mind, let t_0 be the time that the edge is entered, then, the time that the cars exits an edge of length d meters will be T such that:

$$\int_{t_0}^{T} v\,dt = d$$

Which makes, at the end, the weight of the edge $T - t_0$. To find such T, we can apply Barrow's theorem and solve the equation However, since the polynomial is usually of degree 100, this equation can not be solved in the traditional way. Luckily for us, since the function of velocity if strictly positive, I have though of an efficient and precise algorithm that allows us to compute the solution. At first, some observations have to be made; given a T, the result of the integral can easily be computed with a loop, because, let $V(x) = \frac{\alpha_1}{2}x^2 + \frac{\alpha_2}{3}x^3 \dots \frac{\alpha_m}{m+1}x^{m+1} + \beta x$, then, the final integral is given, because of Barrow's rule, by:

$$g(t_0, T) = \int_{t_0}^{T} v\,dt = V(T) - V(t_0)$$

which is a trivial for loop. Moreover, since $v(t)$ is strictly positive, $g(t_0, T)$ will be strictly increasing given a fixed t_0, therefore, there exists one and only one T' such that $g(t_0, T') = d$ which will be our final cost. Consider now the inequality $g(t_0, T) < d$ this will be *true* for all $T < T'$ and *false* for all $T > T'$. We just have to think of an algorithm that can find this point where this inequality changes from *true* to *false*.

To find it, we will set a search range $[low, top]$ defined by two boundaries *low* and *top*. Assume $low = t_0$ at the start and *top* to be big enough such that $top > T'$, then, it is obvious that $T' \in [low, top]$ so we can proceed to search it. Consider $T_1 = \frac{a+b}{2}$, then, if $g(t_0, T_1) < d$, it means that $T' > T_1$ which reduces our search range to $[T_1, top]$, otherwise, if $g(T_1) > d$ it means that $T' > T_1$ which reduces our search range to $[low, T_1]$. Now, repeat this process again, consider $T_2 = \frac{low+T}{2}$ or $\frac{T+top}{2}$ and so on. To stop the process, consider a really small value $\epsilon = 10^{-12}$ and end the iteration when $top - low < epsilon$. By the process followed, this assures us that at the end of the run, $T' \approx low \pm epsilon$ which is quite precise.

It can easily be seen that this algorithm is blazingly fast because it divides the search range by a half in each iteration. It will end up being something of the order $O(\log(x))$. However,

computing $g(x)$ every iteration takes an $O(klog(k))$ complexity (being k the degree of the polynomial) because exponentiation is an $O(log(k))$ algorithm. This makes the generator function $O(klog(k)log(x))$

Algorithm 22 Compute the weight of the edge

```
function edgeWeight(t, g(x), d)
    EPS = 10^-12                                  ▷ Define epsilon
    low ← t                                       ▷ Define low
    top ← low + 10^8                              ▷ Define top
    while top − low > EPS do                      ▷ run until it is precise enough
        T ← (top + low)/2                         ▷ fix T
        if g(t, T) < d then
            low = T                               ▷ reduce search range
        else
            top = T                               ▷ reduce search range
        end if
    end while
    T' ← low
    return T'
end function
```

It can now be said, that given an entering time t_0 to the edge, the cost of the edge will be defined by the mentioned function $(T' - t_0)$. This is, in all its majesty the definition of a non-static graph, an edge that changes its weight depending on a parameter that is begin accumulated during the algorithm. At the end, the generator function of the graph will be defined by the following process[33]:

[33]distanceDataset is the dataset that has the information about the length of each road

$$\begin{aligned} d &= distanceDataset[i][j] \\ ans &= PolynomialFitting(dataset) \\ g(t,y) &= \int_t^y \sum_{i=1}^{m} ans_i x^i \, dx + y \cdot ans_{m+1} - t \cdot ans_{m+1} \\ f_{ij}(t_0) &= edgeWeight(t_0, g, d) - t_0 \end{aligned} \tag{33}$$

Testing the algorithm

To test the algorithm, I will define various arbitrary arrays of coefficients of the polynomial, compute the final value T' with a graphic calculator[34] and compare it with the given T' by the algorithm. Some of the tried *ans* vectors are:

$$\begin{aligned} ans &= \{1,1,3\}; t_0 = 5; d = 4 \rightarrow T \approx 5.11884 \\ ans &= \{0.1,0,3,2,1\}; t_0 = 0; d = 5 \rightarrow T \approx 1.29912 \end{aligned} \tag{34}$$

Although albeit a proof, since all the results are the same as the graphics calculator it can practically be said that the algorithm works. However, there is a case that we have not taken into consideration, if the algorithm is traversing the graph and is currently in a vertex. It can have several options take a long road (full of traffic) or a sequence of fast small ones. Since it is traversing the graph far away from the source or the target, small edges are eliminated by the error function which means that it will not take the skip. To fix this, small roads will be considered if they are in a range R' of a main road.

7.3 Error function

Now that all we can assume that the generator functions well implemented, it is time to define how the error function is going to be. In our case, the error function will only serve a matter of eliminating the edges that seem to not be contributing to the final graph, improving therefore the efficiency of the run algorithm because it has less edges to go through. The question is: What

[34] I will compute the integral by hand, plot it in a graphing calculator and see where it intersects with $y = d$

are the roads that we want to take into consideration when planning long trips across the country?

At first glance, we do not want to evaluate roads that are really small and would differ really much from the direction that we want to go but we can rapidly see that if the trip starts in a big city, leaving it need small roads. To solve this, as it has already been said, for small roads, the traffic is not predicted and the weigh of the edge will be defined as $w(e_{ij}) = \frac{d}{v}$ where v is the average allowed velocity. Nevertheless, we only want to take into account this small roads when we are close to the source and target vertices and main roads[35] The error function has to do this, if the road is in the category of small and not close enough of the source, target or big road, it has to return false, otherwise, true. Moreover, a relation boundary could be defined between the time taken and the length of the road and eliminate the edge if it the cost is bigger than the boundary (which means it is a really slow edge). This would force the algorithm into computing an alternative path without the big traffic jam. At the end, let (x_s, y_s), (x_t, y_t) be the coordinates of the source and target respectively and (x_{ij}, y_{ij}) be the coordinate of the center of the edge e_{ij}, then, if the distance boundary is R, then the first conditions that we have to take into account are:

$$h_{mn}(a,b) = \sqrt{(a - x_{mn})^2 + (b - y_{mn}^2)}$$

$$C_1(e_{ij}) = \begin{cases} 0 \text{ if } h_{ij}(x_s, y_s) > R \text{ and } h_{ij}(x_t, y_t) > R \text{ and road is small} \\ 1 \text{ if } h_{ij}(x_s, y_s) \leq R \text{ or } h_{ij}(x_t, y_t) \leq R \text{ and road is small} \\ 1 \text{ if road is big} \end{cases} \tag{35}$$

Now, we have to take into account the edges that are too slow to traverse. To do so, one extra condition that returns 0 if $d/f_{ij}(x) < k$ where k is a chosen boundary has to be added. This condition basically deletes and edge if the average velocity over all the edge is smaller than k which supposedly means that there is a traffic jam.

$$C_2(e_{ij} = 0 \text{ if } \frac{d_{ij}}{f_{ij}(x)} < k$$

[35]To determine whether a road is a main road we can do it based on the amount of data that we have per week or say that it is a big road if it is larger than a chosen x.

Now, we have two conditions that return 1 or 0 and that overlap between them, however, we want to make an error function such that if at least one condition returns 0, the edge is deleted. To do so, the binary operator and & is going to be used (see Annex A for information about binary operators). This last observation yields us our error function:

$$error(e_{ij}) = C_1(e_{ij}) \ \& \ C_2(e_{ij}) \tag{36}$$

7.4 The algorithm

We already know that to find the shortest path from one point to another one the best algorithm that exists is Dijkstra's algorithm. However, this algorithm has only been proven to work on conventional graphs that remain constant through all the process. Nevertheless, it can be seen that Dijkstra's algorithm will still work on a changing graph but it will be slower and less efficient.

7.4.1 Standing still in a vertex

The first problem comes with the possibility that, whilst traversing the graph, staying put in a vertex and traversing the edge some time after will be more efficient than a continuous travel. If this is true, it will be impossible to find such path because there will be endless possibilities. Thankfully, it can be proven that staying put in a vertex for t_s seconds will never be efficient. Let v_i be the vertex that the algorithm is at, then, it can take any existing edge e_{ij}. The arriving time at the node is t_0, thus, we have to prove, with $t_s > 0$, and $t_x = t_s + t_0$:

$$\int_{t_0}^{T} v_{ij}(t)\,dt = \int_{t_x}^{T'} v_{ij}(t)\,dt = d \Rightarrow T < T' \tag{37}$$

This means that the leaving time given by staying put in a vertex is always greater than a continuous run. The argument is quite simple, assume that $T' < T$, then, because $t_0 < t_x$ and since $v(t)$ is strictly positive, making all the integrals positive:

$$\int_{t_x}^{T'} v(t)\,dt < \int_{t_0}^{t_x} v(t)\,dt + \int_{t_x}^{T} v(t)\,dt$$

Which means that:

$$d < d$$

Which is a contradiction. This assures that $T < T'$ which means that staying put in a vertex is never efficient. With the same argument it can also be seen that is is never efficient to go back through an edge which are the necessary conditions to make Dijkstra's algorithm work.

7.4.2 Process

It is important to remind that the t_0 that we have been taking about all the anterior sections is the entering time of the vertex. Keeping this in mind, the algorithm is really similar to Dijkstra's. We initialize a priority queue with the source vertex and follow the exact same process described in 2.5. The only change will be the relaxation process and the computation of the edge weight. As it has been said, $dist[v_i]$ stores the current length of the shortest path from the source vertex to v_i which will be our t_0. Making the weight of any edge that wants to be evaluated in the relaxation process[36]:

$$w(e_{ij}) = f_{ij}(dist[v_i])$$

Which will make the relaxation operation of a vertex v_j:

$$dist[v_j] := \mathbf{min}(dist[v_j], dist[v_i] + f_{ij}(dist[v_i])$$

All the remaining algorithm is the same as a Dijkstra.

7.4.3 Evaluation of the interaction between the graph and the algorithm

Thanks to using non-static graphs, it can be seen that the algorithm that is being run on the graph is constantly interacting the the graph and changing the configuration of the data structure. The question is "How is this graph changing?".

The graph is going to interact in three ways: the costs of the edges will change, the density of the graph will change (more roads will be evaluated in certain points) and the inefficient roads will not be taken into consideration depending on the state of the algorithm.

[36] $f_{ij}(c)$ is the generator function mentioned in equation 33

The first interaction is the obvious. As we have seen, the generator function takes as parameter t_0 which is the entry time of the edge. Depending on this t_0 the cost of the edge will be different. This means that if the algorithm gets to the vertex v_i with an accumulated time of t_1, the adjacent vertices and the cost of the adjacent edges will be different than if it had arrived to the same vertex with an accumulated time of $t_2 \neq t_1$.

Moreover it will change the density of the roads depending on the distance from the source or target that the algorithm is at. As it has been seen, because when we are close to the source we want to evaluate the graph just like in the case mentioned in section 7.2.1 the graph, thanks to the error function will change to a more dense and precise graph, which will lead to better results.

The last interaction is that inefficient roads, which have a lot of traffic will be deleted. Since this condition depends on the generator function which itself also depends on the parameter t_0, this condition interacts with the algorithm because depending on the entry time t_0 the edge might not exist.

7.4.4 Complexity

As it is basically a Dijkstra, it has an already proven complexity of the order of $nlog(n)$ for sparse graphs, which thanks to the error function it is our case because it will eliminate all the necessary edges. However, the relaxation process requires the generator function which takes $O(log(x))$ to search for T and $O(klog(k))$ being k the degree of the polynomial to compute the integral each search. Which as it has been said makes the generator function of a total time complexity of $O(klog(k)log(x))$. This leaves the total complexity of this new approach to the algorithm as $O(nlog(n)klog(k)log(x))$ which can be approximated as $O(n^2 \log^3(n))$ if the degree of the polynomial is high. All of this is taking into account that the machine learning algorithms are run as a pre computation meaning that at the start of the algorithm all the polynomials coefficients are already computed and all the data has been process. Had not been this the case, we would have to add the complexity of the machine learning to the algorithm complexity.

7.5 Final results

As the final results, we have obtained a graph that interacts with the algorithms that is running with it. This has been accomplished because as the Dijkstra traverses the graph, it accumulates weight (time). This weight is then used to determine the cost of the generator function because it takes as a parameter the starting time t_0. This means that getting to a vertex v_i from a path ρ_1 or a path ρ_2 can result into completely different graphs if the accumulated time of the first path is different from the second one.

On the wrong side, I have not been able the prove that the algorithm works on changing edges, however, I have proved that it is not optimal to stay put on an edge. After proving this, it can be seen intuitively that albeit the algorithm might take a little bit longer, it still will find the correct path because it will still go though all possibilities in the same order as Dijkstra and this one has been proven to work. However, even though this algorithm might at the end be an heuristic, it works nearly every time.

In this last image of the article I will show a 3D image of the different phases that a graph goes through whilst the Dijkstra runs on it. It is similar to the 3D representation of a non-static graph mentioned previously, but follows different rules. In the x ans y axis, the configuration of the graph is drawn, however, on the z axis, for every edge traverse by the Dijkstra, the graph configuration is drawn with the accumulated weight of the running Dijkstra. Overall, it is a mere simulation of the state of the graph if the Dijkstra algorithm is evaluating the edge with $t_0 = z$.

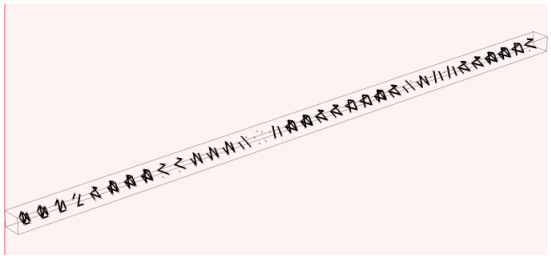

Figure 33: Dijkstra graph

8 Conclusions and ending

We have started the monograph with no knowledge about that beautiful theory which are graphs. We have seen how the first graphs were build and gone from really simple theorems to more complex ones, rediscovering what mathematicians proved centuries ago. This has made me realise the complexity of such simple looking topic and the precision in which a basic structure of lines and dots can abstractly represent situations of the real world. Moreover, we have gone though the implementation of the most basic traversal algorithms which, without any doubt, has made me thing highly of the ones that thought about them for the fist time.

After the brief introduction, we have seen the birth of what can potentially be a new way to see graphs, we have put complexity without limits to a really simple idea of connections between nodes. Using machine learning and statistic predictions, we have managed to create a graph that interacts with the algorithms that are being run on it and that can more accurately represent real life situations.

To end it we have found two interesting applications of the discussed above, proving once and for all the usefulness of such theory.

8.1 Traditional Graphs

As traditional graphs concerns I can say that all the proposed objectives have been met. I have managed to rediscover graphs from the perspective of mathematics and proven, making use of abstract mathematics, the most important properties of graphs.

Moreover, from a computational stand point, it has also been a success. All the algorithms mentioned in the firs part of the monographs have been implemented efficiently and work to near perfection. It is true that I would have liked to put a little more emphasis on the meeting point of mathematics and computer science: the computational analysis of the properties that mathematics gives us.

As difficulties, I had a little bit of trouble explaining the process of the algorithm because they are very abstract, but overall, I am pleased with the first part of the research.

8.2 Non-static graphs

As the mathematical definition of a none-static graphs, all the objectives have also been met. The main question that has risen at the beginning of the project has successfully been answered: "What happens if the weight of a graph is a function instead of a constant?". Moreover, all the concepts and ideas that followed this question have been formally put in paper and I have successfully managed to define this new type of graphs that had risen as a consequence, making this banal question a beginning of what could be a complete new way of interpreting graphs.

The second part of non-static graphs, where I apply machine learning to predict the generator function, has not been as successful as I would have liked. On one side, I have managed to precisely describe the process and to go through all the theoretical part about it, however, I would have like to use a more general machine learning algorithm which lead to more accurate predictions. On the bright side, the algorithms described work for the specific cases mentioned and, overall, the idea behind it is quite clear. To add more, the polynomial that the polynomial fitting yields is often of degree 70 for small datasets which nearly every time leads to over-fitting if the dataset does not exactly follow a polynomial tendency. This, as I have said, could be fixed with other machine learning algorithms that produce more accurate results, but I do not have neither the required datasets nor the ability to program those.

On the other hand, computationally speaking, this part of the research has been a total success, all the algorithms mentioned in the section have been implemented without any difficulties and thoroughly tested with computer generated datasets.

8.3 Connecting Cities

This minor section had been done as how traditional graphs could be easily applied to the real world without much difficulty. On one hand I like the final result because the algorithm works perfectly and even though it will not make a difference it can help readers realise that small

actions can change the world. On the other hand, I have the feeling that this section falls a little bit out of topic but however the case, it is not really long to read and is a practical application of the error function for non-static trees that brings together all the learned about trees in the first section.

8.4 Shortest path with traffic

This last section is the point where all the maths and formal definitions of the new concept of graphs come together and it can be said that the results are promising. Even though it is one of the most mundane applications of non-static graphs, the simplicity about it and the fact that I reused Dijkstra's algorithm make this application really interesting. On the bright side, it can be said that the algorithm works and that the idea has been well implemented. However, since nowadays data is really private and difficult to come by, specially for concrete things like mine, I have not managed to test the algorithm on a real case which is unfortunate because the hours of research for a dataset have not paid off. Moreover, since the polynomial fitting algorithm is really concrete, the predictions does not quite work so the generator functions are not well defined.

8.5 How to further on the investigation

To further on the investigation it could be great to keep on developing the idea of non-static graphs. I have just defined that basis of them and I feel that they have much more potential than the one that I have been giving them on this essay. For example, an idea that comes into my mind, is modelling big molecules such as proteins as graphs where each atom is a node with some spacial coordinates (x, y, z) and the edges are bonds. Then, set the weights of the edges with some function that is calculated with machine learning that predicts the stability of this edge existing. To predict the stability the machine learning algorithms learns form 3D structures that have already been computed experimentally. If it is very likely that this connection happens, it is stable, the weight is small, otherwise huge. Then, set an recursive error function that given a configuration of the graph, modifies the edges (eliminates and replaces them) in such a way to improve stability. This would lead to a graph that feeds itself and evolves with time. The algorithm is left running for some hours and in the end it should show the 3D structure of the predicted molecule. However, this is just a mundane idea that I had but was too hard to develop.

To add more, a nice way to continue the research one non-static graphs would be finding a way to get a real dataset of a real situation and trying to use it to make a real project about non-static graphs. However, right now I do not have to tools to do so.

Another way to keep developing the theory is trying to use more efficient and precise machine learning algorithms to find the generator functions such as Deep learning or CNN.

The last comment on how to improve the research is that it would be great to find other algorithms that might work with non-static graphs or define new algorithms that only work with these types of graphs.

8.6 Ending

To finish, I would like to point out that this is just an introduction of an idea I had, it is by no means an intention to close the topic and anyone that would like to further the idea is welcome to do so.

Thanks for reading through all the article.

References

[AAA59] SHELDON B AKERS, NORMAN N ALPERIN, and CRAIG ANDREWS. International symposium on the. In *Proceedings*, volume 29. Harvard University Press, 1959.

[D+59] Edsger W Dijkstra et al. A note on two problems in connexion with graphs. *Numerische mathematik*, 1(1):269–271, 1959.

[Fie73] Miroslav Fiedler. Algebraic connectivity of graphs. *Czechoslovak mathematical journal*, 23(2):298–305, 1973.

[Joy16] David Joyce. Math 218–topics in statistics. 2016.

[KDT03] KM Koh, FM Dong, and EG Tay. Graphs and their applications (3). *Mathematical Medley*, 30(1):11–22, 2003.

[Kru56] Joseph B Kruskal. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proceedings of the American Mathematical society*, 7(1):48–50, 1956.

[KWW22] Mykel J Kochenderfer, Tim A Wheeler, and Kyle H Wray. *Algorithms for decision making*. Mit Press, 2022.

[LG+12] Olivier Lézoray, Leo Grady, et al. Graph theory concepts and definitions used in image processing and analysis. *Image Processing and Analysis*, pages 1–63, 2012.

[RN02] Stuart Russell and Peter Norvig. Artificial intelligence: a modern approach. 2002.

[Sal07] Jimmy Salvatore. Bipartite graphs and problem solving. *Univ Chicago*, pages 1–7, 2007.

[Sko15] Nolan Skochdopole. Discrete mathematics and algorithms icme refresher course. 2015.

[SV08] Alex Smola and SVN Vishwanathan. Introduction to machine learning. *Cambridge University, UK*, 32(34):2008, 2008.

[Wal79] WD Wallis. Graph theory with applications (ja bondy and usr murty), 1979.

[Wol91] Stephen Wolfram. *Mathematica: a system for doing mathematics by computer.* Addison Wesley Longman Publishing Co., Inc., 1991.

"Everything has an ending . Make yourself at peace with that and everything will be well"

Jack Korfield

This page has been intentionally left blank

9 Annex A, Further information

The aim of Annex A is to make some clarifications about some concepts mentioned though the article, further on some topics that might help with the understanding of the project, and finally comment and present all the code that has been appearing though the text. Annex A is totally optional to read, and is only here to help with the total understanding of the project.

9.1 Big O notation

In programming there are many cases in which we would like to compare two algorithms. To do so, we define the efficiency of an algorithm as the time or number of steps taken to solve an input of size n. For example, when analysing an algorithm, one might find that the number of steps to solve run it on an input of size n is given by $T(n) = 2n^2 + \frac{1}{2}n + 1$. Ignoring the lower order members, it can be said that the algorithm has an order of n^2 so it can be written as $T(n) = n^2$. To express the speed in which the algorithm solves the problem, we will use the $O(T(n))$ notation. It is important to note that when finding the efficiency of an algorithm, the worst case is always assumed. For example, if a naive algorithm is designed to find the shortest path from vertex v_i to v_j trying all such possible paths, it will be assumed that the correct path is the last one tried. Having stated this, we formally define $O(f)$ of a given function f as:

$$O(f) = \{g(x) | \exists A, a \in \mathbb{R}^+ \text{such that } |g(x)| \leq A|f(x)| \; \forall \, x > a\}$$

or

$$O(f) = \{g(x) | \lim_{x \to \infty} \left(\frac{g(x)}{f(x)}\right) < \infty\}$$

To put it differently, $O(f)$ are all functions that are of the same or less order than f. This also means that if an arbitrary algorithm A with its time function $T_A(n)$, runs in less than t seconds for our input size, all other algorithms B whose time functions $T_B(n) \in O(T_A(n))$ will perform similarly. Having seen this we can already see the usefulness of this notation, it is possible to predict whether an algorithm will work for our input size before implementing it, to do so, we just have to mathematically calculate its time function, which in most cases is not hard.

9.2 Binary Operators

Binary operators are operators that deal with numbers in their binary form. To work through the four operators, we will use two arbitrary numbers a and b and for the examples 2 eight bit numbers.

The five main bitwise operators are:

$a \ll b$ *Shift left* This operator will move all the binary digits of a, b positions to the left. For example $5 \ll 2 = 20 \rightarrow 00000101 \ll 2 = 00010100$

$a \gg b$ *Shift right* This operator will move all the binary digits of a, b positions to the right. $5 \gg 2 = 1 \rightarrow 00000101 \gg 2 = 00000001$

$a \wedge b$ *And* And operator will compare each position of a with the corresponding position on b, and will leave 1 if both are 1 and 0 otherwise. $10001101 \wedge 11101100 = 10001100$

$a \vee b$ *Or* Or operator will also iterate through every position of a and compare it with the corresponding one in b. It will leave 1 if at least one of the digits is 1 and 0 otherwise. $10001101 \vee 11101100 = 1111101$

$a \oplus b$ *Xor* The excluding or operator will also compare each bit of a with the corresponding bit of b and will return 1 with both bits differ and 0 if the bits are equal. $10001101 \oplus 11101100 = 01110011$

9.3 Bipartite graphs, additional information

A bipartite graph[37] is a connected undirected graph $G = (V, E)$ that can be split into two subsets V_1,V_2 where $V_1 \cap V_2 = 0$ and $V_1 \cup V_2 = V$, such that if $v_i \in V_1$, it is only adjacent to vertices in V_2 and if $v_i \in V_2$, it is only adjacent to vertices in V_1. This means that we can mark every vertex with either 1 or 2, so that no vertex is adjacent to a vertex marked with the same color.

[37] Bipartite graphs will be the base of my interpretation in the last section.

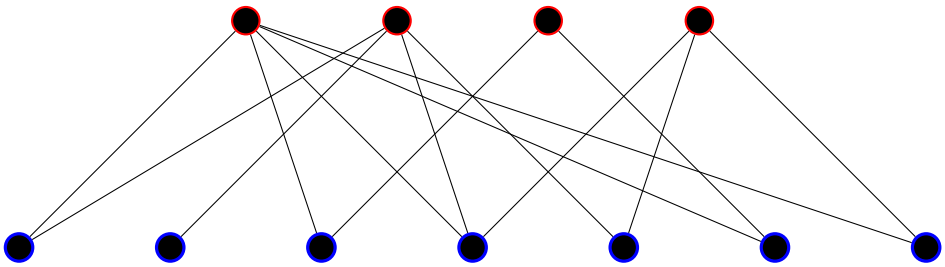

Figure 34: bipartite graph G_E

Property 1: *In a bipartite graph, a path $\rho(v_i, v_j)$ will always intercalate vertices from V_1 and V_2.*

Proof: This property is deduced from the mere definition of itself. Let v_k be an arbitrarily chosen vertex from $\rho(v_i, v_j)$. If $v_k \in V_1$, by definition, since it is only adjacent to vertices in V_2, v_{k+1} will be in V_2, otherwise, if $v_k \in V_2$, the next vertex in the path will be in V_1.

Property 2: *Let V_1 and V_2 be the bipartitions of size n_1,n_2 of the graph G, then, $\sum_{v_i \in V_1} \delta(v_i) = \sum_{v_j \in V_2} \delta(v_j)$.*

Proof: We will prove it using induction. The base case where $|E| = 1$ is trivial, since the edge has to connect a vertex v_1 from V_1 to a vertex v_2 from V_2, therefore, $\delta(v_1) = \delta(v_2) = 1$, thus the property holds.

We assume that for $|E| = n$ the equality holds. Consider now a bipartite graph G of $|E| = n+1$, by the induction hypothesis, $\sum_{v_i \in V_1} \delta(v_i) = \sum_{v_j \in V_2} \delta(v_j)$. Now, delete an edge, since this edge will be connected to both a vertex from V_1 and a vertex from V_2, it will subtract 1 from both $\sum_{v_i \in V_1} \delta(v_i)$ and $\sum_{v_j \in V_2} \delta(v_j)$, hence the property holds, and by the induction hypothesis, it has been proved.

Property 3: *A bipartite graph contains no odd length cycles.*

Proof: We will prove this identity by contradiction. Let G be a bipartite graph with n vertices and m edges. Now, assume there is a cycle $C_{2k+1} = v_1, e_1, \ldots, v_{2k+1}, e_{2k+1}, v_1$. Because of property 1, v_{2k+1} has to be in the same partition as v_1, therefore, since by definition, two edges in the same partition are not adjacent, the edge connecting v_1 and v_{k+1} will not exist. Thus, by

contradiction, an odd length cycle is impossible.

Having stated these basic properties, I will proceed to define a concrete type of bipartite graph. We say that graph is a complete bipartite if $\exists e_{ij} \in E_1 \forall v_i \in V_1, v_j \in V_2$. In other words, if there is a connection between each vertex from a partition to every vertex from the other partition. Since every v_i will generate $|V_2|$ edges, in complete bipartite graphs $|E| = |V_1| \cdot |V_2|$.

Algorithm 23 Stating if a given graph is bipartite of not

Input: $\{e_1, e_2, \ldots, e_{n-1}, e_n\}$

```
Declare: mark,components,adj
functionDFS(v_i, markId
    mark[v_i]← markId
    bipartite ←true
    for j in adj[v_i] do
        if mark[v_j] = -1 then
            bipartite ← bipartite ∧ DFS(v_j, markId ⊗1)   ▷ Goes to v_j with the opposite mark
        else
            if markId = mark[v_j] then                    ▷ It encounters the same mark
                bipartite ← false                         ▷ It is no longer bipartite
            end if
        end if
    end for
    return bipartite
end function
```

Output: The function will return *true* if the graph is bipartite and *false* otherwise.

The function written on Algorithm 23 will return *true* if the given adjacency list corresponds to a bipartite graph and *false* otherwise. To see what $\otimes$ and $\wedge$ stand for, see Annex.

9.4 An interesting arithmetic proof using complete graphs

Graph theory does not just stay on graphs, it can be applicable to other fields of maths, such as combinatorics or arithmetic. For example, a complete graph can be used to prove that $0 \leq k \leq n$, $\binom{n}{2} = \binom{k}{2} + k(n-k) + \binom{n-k}{2}$. (we will use a double counting) To do so, we just take two complete graphs (see figure 35) K_k and K_{n-k}. We trivially notice that the sum of vertices is n, so if both graphs were a single one and it was complete, it would have $\binom{n}{2}$ edges. Having stated that, we know that K_k will have $\binom{k}{2}$ edges and K_{n-k} $\binom{n-k}{2}$ so we can write that

$$\binom{n}{2} = \binom{k}{2} + \binom{n-k}{2} + \alpha$$

We also know that the remaining edges to complete the union of both graphs are $k(n-k)$ as we have to connect each k vertices of K_k with every $(n-k)$ vertices of K_{n-k}. Finally we have that:

$$\binom{n}{2} = \binom{k}{2} + \binom{n-k}{2} + k(n-k)$$

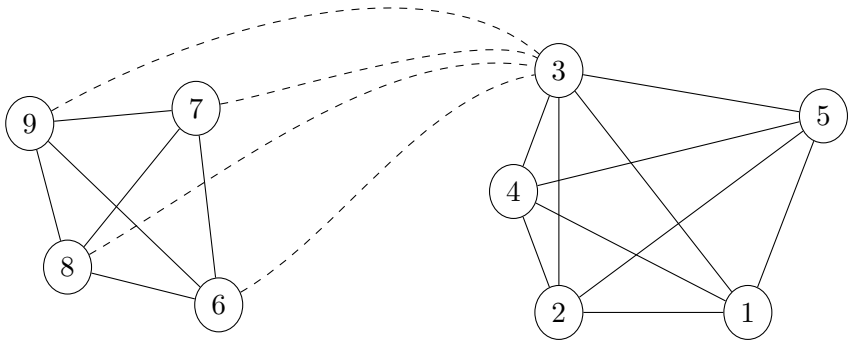

Figure 35: Illustration of the proof

10 Annex B, used applications

For the development of this project I have used numerous free source programs that are worthy of a mention in this last Annex. Most of them are related to the coding aspect, the visual and the generation of all the images that are put through the monograph.

10.1 LaTeX

Latex is a free source programming language that compiles to Pdf that allows people to formally write a math paper without any complication. This entire paper has been developed with latex from start to the end and without any doubt had I not had access to LaTeX, i would not have been able to pull off this result. The facility in which LaTeXhas allowed me to write all the complicated math signs is outstanding to say the least and I am grateful that such program exists. I would completely recommend any math student or even high school students to learn it because not only has it helped me write this paper but now I write all the science assignments with this program (specially Physics and Chemistry).

10.2 Wolfram Mathematica

Wolfram Mathematica has paid a crucial role in the production of the three dimensional images of non-static graphs. Mathematica is a programming language specialised in mathematics and has basically every functionality that a mathematician would ever use. Despite only using a glimpse of the program I can say that it offers endless graphing possibilities and brigs mathematics to another level.

To produce the images, since I am note really familiar with the language I have added a method called print in the C++non-static graphs class which writes by itself the Mathematica code that produces the graph in a given concrete configuration.

At the end, I would also recommend anyone to get a hand on Mathematica, even if it is only the basics.

10.3 Tikzpicture

Tikzpicture is a library of LaTeXand has been used to generate all the two dimensional graph images that are present during all the article. It allows to write latex code that compiles to images which is not possible with plain latex. Even if it is very niche, the are lots of image latex libraries but I have found this one to be the most efficient one.

10.4 Desmos

Desmos is an online graphing calculator that I have used a little bit to visualise functions whilst producing non-static graphs, testing the machine learning and producing three of the function images that can be seen in section 3. Even though all this could have been done with Mathematica I used this on-line graphing calculator because of its practicality and easy access.

10.5 C++

This could not end without a proper mention to the C++language because without it not even the possibility of doing this research could have been possible. All the mentioned algorithms have been implemented with C++and thoroughly tested with it.