Graduado en Matemáticas e Informática

Universidad Politécnica de Madrid

Escuela Técnica Superior de Ingenieros Informáticos

TRABAJO FIN DE GRADO

GTapp: Dominating Sets within Triangulations

Autor: Héctor Moreno Cervera

Director: Gregorio Hernández Peñalver

MADRID, JUNIO 2016
# Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Resumen</td>
<td>vii</td>
</tr>
<tr>
<td>Summary</td>
<td>viii</td>
</tr>
<tr>
<td>1 Introduction</td>
<td>1</td>
</tr>
<tr>
<td>1.1 Motivation</td>
<td>1</td>
</tr>
<tr>
<td>1.2 Project scope</td>
<td>1</td>
</tr>
<tr>
<td>1.3 Portfolio structure</td>
<td>2</td>
</tr>
<tr>
<td>2 Mathematical ground</td>
<td>3</td>
</tr>
<tr>
<td>2.1 Graph theory</td>
<td>3</td>
</tr>
<tr>
<td>2.2 Triangulation</td>
<td>5</td>
</tr>
<tr>
<td>2.2.1 Convex Hull</td>
<td>6</td>
</tr>
<tr>
<td>2.2.2 Triangulation strategies</td>
<td>7</td>
</tr>
<tr>
<td>2.3 Dominating set</td>
<td>7</td>
</tr>
<tr>
<td>2.3.1 Dominating set restrictions</td>
<td>8</td>
</tr>
<tr>
<td>3 Computer science ground</td>
<td>10</td>
</tr>
<tr>
<td>3.1 Structured programming</td>
<td>10</td>
</tr>
<tr>
<td>3.2 Object oriented programming</td>
<td>11</td>
</tr>
<tr>
<td>3.3 Event driven programming</td>
<td>11</td>
</tr>
<tr>
<td>3.4 Language and IDE</td>
<td>13</td>
</tr>
<tr>
<td>4 GTapp</td>
<td>15</td>
</tr>
<tr>
<td>4.1 Program Structure</td>
<td>15</td>
</tr>
<tr>
<td>4.1.1 Configuration</td>
<td>16</td>
</tr>
<tr>
<td>4.1.2 Graph</td>
<td>17</td>
</tr>
<tr>
<td>4.1.3 Dominations</td>
<td>17</td>
</tr>
<tr>
<td>4.1.4 Triangulations</td>
<td>18</td>
</tr>
<tr>
<td>4.1.5 Window</td>
<td>18</td>
</tr>
<tr>
<td>4.2 Painting execution flow</td>
<td>23</td>
</tr>
<tr>
<td>4.3 Triangulation algorithms</td>
<td>27</td>
</tr>
<tr>
<td>4.3.1 Convex hull</td>
<td>28</td>
</tr>
<tr>
<td>4.3.2 Layered triangulation</td>
<td>29</td>
</tr>
<tr>
<td>4.3.3 Random triangulation</td>
<td>29</td>
</tr>
<tr>
<td>4.4 Dominating set algorithms</td>
<td>30</td>
</tr>
<tr>
<td>4.4.1 Greedy algorithm</td>
<td>30</td>
</tr>
<tr>
<td>4.4.2 Connected algorithm</td>
<td>32</td>
</tr>
<tr>
<td>Section</td>
<td>Title</td>
</tr>
<tr>
<td>---------</td>
<td>-------</td>
</tr>
<tr>
<td>4.4.3</td>
<td>Independant algorithm</td>
</tr>
<tr>
<td>4.4.4</td>
<td>Total algorithm</td>
</tr>
<tr>
<td>4.4.5</td>
<td>Paired algorithm</td>
</tr>
<tr>
<td>4.4.6</td>
<td>Distance 2 algorithm</td>
</tr>
<tr>
<td>5</td>
<td>Experimental research</td>
</tr>
<tr>
<td>5.1</td>
<td>Storing data</td>
</tr>
<tr>
<td>5.2</td>
<td>Data mapping</td>
</tr>
<tr>
<td>5.2.1</td>
<td>D/V Graph</td>
</tr>
<tr>
<td>5.2.2</td>
<td>D/K Graph</td>
</tr>
<tr>
<td>6</td>
<td>Conclusion</td>
</tr>
<tr>
<td>7</td>
<td>Bibliography</td>
</tr>
<tr>
<td>A</td>
<td>GTapp file architecture</td>
</tr>
<tr>
<td>B</td>
<td>Triangulations Source Code</td>
</tr>
<tr>
<td>B.1</td>
<td>Convex Hull Algorithm</td>
</tr>
<tr>
<td>B.2</td>
<td>Layered Triangulation Algorithm</td>
</tr>
<tr>
<td>B.3</td>
<td>Random Triangulation Algorithm</td>
</tr>
<tr>
<td>C</td>
<td>Dominating set Source Code</td>
</tr>
<tr>
<td>C.1</td>
<td>Greedy algorithm</td>
</tr>
<tr>
<td>C.2</td>
<td>Connected algorithm</td>
</tr>
<tr>
<td>C.3</td>
<td>Independant algorithm</td>
</tr>
<tr>
<td>D</td>
<td>Experimental Results</td>
</tr>
<tr>
<td>D.1</td>
<td>Excel programming</td>
</tr>
<tr>
<td>D.2</td>
<td>Linear Regression</td>
</tr>
<tr>
<td>D.2.1</td>
<td>$D$ over $V$</td>
</tr>
<tr>
<td>D.2.2</td>
<td>$D$ over $K$</td>
</tr>
</tbody>
</table>
Resumen

El incontrolado desarrollo de las ciencias de la computación en las últimas décadas ha provocado que actualmente se generen millones de bytes de contenido cada minuto a lo largo del globo. Es por esto que la teoría de grafos, rama dentro de las matemáticas discretas, cobra cada vez más importancia dentro de las ciencias de la información. Esta teoría no solo nos permite modelar grafos, sino que también permite obtener información a partir de redes de nodos. La confluencia de las matemáticas con la informática en este ámbito es lo que hace tan atractivo este proyecto como trabajo de fin de grado.

A lo largo de la presente memoria estudiaremos las triangulaciones de grafos, comúnmente utilizadas con el objetivo de modelizar redes inalámbricas, y los conjuntos dominantes, que proporcionan conjuntos de vértices característicos. Este estudio está orientado bajo un punto de vista experimental, donde buscamos obtener resultados a partir datos estadísticos gracias a muestras aleatorias de vértices. Con el fin de conseguir dichas muestras, se ha desarrollado una aplicación que permite visualizar grafos y generar datos estadísticos relativos a los conjuntos dominantes y triangulaciones implementadas. Además, se han obtenido resultados que han permitido establecer nuevas cotas experimentales, además de comprender en mejor medida cuán de buenos eran los diferentes algoritmos propuestos.
Summary

The uncontrolled computer science development over the past decades has lead us to a world where millions of unique bytes of information are being generated every minute. That is the reason behind why graph theory, field which lies within the discrete mathematics, has became increasingly significant for information technologies. This theory not only implies modeling graphs, but also allows obtaining information based on connected networks. The confluence between mathematics and computer science in this field is what makes this project attractive as a final degree portfolio.

Along this paper we will study graph triangulations, commonly used for modeling wireless networks, and dominating sets, which provide sets of characteristic nodes. This research is oriented under an experimental point of view, where we try to obtain results upon statistical data thanks to randomly distributed node samples. With the purpose of achieving this samples, we have developed an application capable of visualizing graphs and generating statistical data regarding the implemented dominating sets and triangulations. Moreover, acquired results have permitted establishing new experimental boundaries, as well as led us to understand which algorithm performed better among the proposed.
Chapter 1

Introduction

1.1 Motivation

Present-day technological improvements have inevitably forced a better understanding of how large volumes of content are related. The computer science development over the last decades, greatly influenced by the Internet, has lead us to a world where millions of unique bytes of information are being generated every minute [8]. Due to the ungovernable knowledge available, computer scientists are trying to obtain meta-information from this large sets of related databases.

In their pursue for data analysis techniques, computer sciences have found a rather dormant field in mathematics. The graph theory, which was initially conceived as a field for studying relations between objects, has become increasingly significant because of its capability to model computer networks. Several conceptions as for instance big data or social networks substantially rely on graph modeling as well as graph related properties.

The current informatics line seems to keep encouraging mathematicians to work in the field of graph theory for the years to come in order to improve how data is modeled, understood and how knowledge can be obtained from big data. One of the many ways how graph theory helps network comprehension is thanks to the concepts of triangulations and dominating sets.

On one hand, triangulations are known to model wireless networks, where a signal must go through several nodes in order to reach a determined target. On the other hand, connected dominating sets obtain "good" paths to traverse the network. Moreover, different dominating set strategies can result into diverse attractive set of nodes in the network.

1.2 Project scope

The main objective of the present project is to study dominating sets under triangulated graphs. More specifically, we looked at six kinds of dominating strategies and two types of graph triangulations.
We took an experimental approach to this problem, trying to obtain results by generating graphs and gathering knowledge statistically. For this purpose, we implemented a GUI\textsuperscript{1} program named \textit{GTapp}, which stands for \textit{Graph Theory application}. Furthermore as a side goal, we tried to create a highly user friendly scalable application, which could be used by teachers and students to work around the mentioned concepts, not only by visualizing graphs but also by being able to obtain new knowledge.

Therefore, by both obtaining mathematical results and implementing a computing tool, we achieved a final portfolio that manages to cover the fields of mathematics and computer science.

1.3 Portfolio structure

This paper is divided into a fundamentally theoric part (chapters 2 and 3) and a second section refering to the application itself and the obtained results (chapters 4 and 5)

Chapter 2 revolves around the mathematical concepts mentioned throughout the project, attending specifically to the studied triangulations and dominations.

In chapter 3 we dig into the computer science approach and the paradigms involved in the making of the graph theory application. Besides we also cover the language and developing environment involved.

Chapter 4 describes the inner insights of \textit{GTapp}, first by looking at the program structure and later describing the algorithms used for triangulating graphs and obtaining dominating sets.

Meanwhile, chapter 5 presents the experimental results obtained on dominating sets within triangulated graphs thanks to the implemented tool.

Lastly, in chapter 6 we withdraw conclusions from the overall experiences of the assignment.

\textsuperscript{1}Guided User Interface.
Chapter 2
Mathematical ground

2.1 Graph theory

Graph theory lies within the discrete mathematics, the field which studies finite and countable infinite sets. Closely related to computer science, graphs grant us the possibility to model, analyze, simulate and structure data, as well as design practical algorithms.

Maybe because it is widely used nowadays, for instance in computer networks, most people usually forget that this mathematical field emerged in 1736, when Leonard Euler published an article solving the Seven Bridges of Königsberg problem [23]. The exercise is based on the homonymous city, which is run across by the Pregel River, surrounding the Kneiphof island, thus creating four land regions connected by seven bridges [18], as in figure 2.1. Euler solved the problem concluding that it was not possible to go through each of the seven bridges and returning to the starting point without crossing a bridge at least twice [26]. In order to obtain this conclusion, the mathematician modeled the city and the bridges as a graph.

Figure 2.1: Pictures showing how the Seven Bridges of Königsberg are modeled.

The intuitive process of modeling the map of the city of Königsberg serves us as a way for introducing the concept of graph. In this case the land regions (vertex), are connected by bridges (edges).

Definition: A graph G is made up of a set of nodes V and a set of edges E. Each edge connects a pair of nodes. $G = \{V, E\}$
Other characterizations arise upon this definition (e.g. directed graph, multigraph, weighted graph). However, we will not go through these types of graphs because they are not necessary for understanding the present portfolio. Nevertheless, we encourage the reader to investigate different kinds of graphs to better understand the possibilities brought by graph theory.

Now we are in a position from which we can introduce some key concepts often mentioned throughout this paper.

**Definition:** The *degree* of a vertex \( v \), noted as \( \delta(v) \), is the number of edges that connect to it. It can also be understood as the number of vertex related to \( v \).

**Definition:** A *cycle* \( C \) is defined as a sequence of edges starting and ending at the same vertex.

**Definition:** A graph \( G \) is *planar* if it can be represented in the plane in such a way that its edges do not intersect. Contrariwise, a graph said to be *nonplanar* if it can not be represented in a plane without at least two edges intersecting.

**Definition:** In planar graphs, *faces* are the polygons bounded by edges.

**Definition:** A *path* \( p \) is a sequence of edges which connect a sequence of vertex.

**Definition:** A graph \( G \) is *connected* if there is at least one path between every pair of vertex. Contrariwise, a graph is *disconnected* if there is not a path in \( G \) between at least a pair of vertex.

**Definition:** A *connected component* is a subgraph in which any two vertex are connected by a path.

**Corollary:** Connected graphs will be made up of one only connected component.

**Definition:** An *independent set* of a graph \( G \) is a set of vertex where none of them are adjacent\(^1\). Two nodes are independent if there is not an edge connecting them.

**Definition:** The *adjacency matrix* of a graph is a boolean bidimensional list describing which vertex are connected by an edge. The following example (figure 2.3) provides a graph and its adjacency matrix.

\(^1\)Note that a path in \( G \) can exist between a pair of nodes in the independent set.
2.2 Triangulation

In order to understand the concept of a graph triangulation, let’s just consider a graph without edges. Hence, our graph will only be made up of nodes, and we could fill it up with edges by adding them between nodes arbitrarily. Instead, by adding edges so that the inner faces of the graph are three-sided polygons, we can form a triangulation. Thus, the formal definition of a triangulation is the following.

**Definition:** A planar graph $G$ is said to be *triangulated* if the addition of any edge would result into a nonplanar graph.

**Corollary:** A maximal planar graph $G$ is also known as triangulation.

This definition allows a same set of nodes to be triangulated by different sets of edges, which allows us to determine a wide range of triangulation strategies. Each strategy will produce a set of edges that will triangulate the set of nodes which make up a particular triangulated graph. The following figure shows how different triangulations can be obtained from the same set of nodes.

![Graph with nodes and edges](image)

Figure 2.3: Example of two different triangulations built upon the same set of nodes.
2.2.1 Convex Hull

Without regard to which strategy was used for triangulating a particular set of nodes, every possible triangulation will always contain a special set of edges. These are known as the convex hull or convex envelope.

The convex hull $K$ of a graph $G$ is defined as the (smallest) cycle that contains all the nodes in $G$. Therefore, every vertex will either be connected by an edge in $K$ or will belong to the region determined by the convex hull.

As stated at the beginning of this subsection 2.2.1, the convex hull $K$ of a graph $G$ will always belong to any possible triangulation $T$. This can be proven by contradiction considering that if an edge in $K$ does not belong to $T$, then $T$ is not a maximal planar graph and thus not a triangulation of $G$. Figure 2.5 supports the previous demonstration.
2.2.2 Triangulation strategies

For this project we implemented two algorithms for computing different triangulation strategies.

- **Layered triangulation**
  
  Based upon the previous notion, the layered triangulation consists on calculating progressively disjoint convex hulls. The strategy starts by calculating the convex hull of the whole graph, then calculates the hull of the inner nodes, and the process is repeated until there is no convex hull left to be obtained. Afterwards, the triangulation is concluded by connecting the nodes belonging to the subsequent layers, preserving the graph planarity.

- **Random triangulation**
  
  Despite the concept of random triangulation may involve not having a particular strategy, here we describe how we triangulated graphs randomly. For that, first the convex hull is calculated and afterwards the complete triangulation is obtained by adding edges between randomly chosen nodes until the maximum planarity is achieved.

2.3 Dominating set

Whereas the triangulation concept is based on the edges of a graph, the dominating notion revolves around the nodes. In order to understand dominating sets, first we need to introduce the idea of neighbourhood.

**Definition:** The neighbourhood of a node \( \text{neigh}(v) \) is the set of nodes connected by an edge to \( v \).

**Corollary:** If two nodes are connected by an edge, they are said to be neighbours.

Advancing with the preliminar concepts, a dominating set is made up of dominating vertex. This vertex are said to dominate their neighbours. In figure 2.6 it can be seen how node number 2 dominates his neighbours \{1, 3, 7\}.

![Figure 2.6: Example illustrating how a node dominates its neighbours.](image)
The definition of dominating set is achieved by extending this notion to the whole graph. A dominating set $D$ of a graph $G$ is a set of vertex which dominate all the nodes in $G$. The following figure 2.7 shows how for the previous graph, not only $\{2, 6\}$ can be considered as a dominating set, but also $\{1, 2\}$. Although not restricted by the definition, we often want to achieve minimal dominating sets. This would avoid considering $\{1, 2, 6, 7\}$ or even $N = \{1, 2, 3, 4, 5, 6, 7\}$ as valid dominating sets, as in figure 2.8

![Figure 2.7: Example of two different minimal dominating sets.](image)

![Figure 2.8: Example of two over populated dominating sets.](image)

### 2.3.1 Dominating set restrictions

Similar to how we could build different types of triangulations, it is possible to obtain distinct dominating sets according to certain restrictions. Ahead we define each of the possible dominations considered in the project\(^2\).

**Definition:** Let $G = (V, E)$ be a graph. A subset $S$ of $V$ is called dominating if every vertex $v$ is either contained in $S$ or a neighbour of an element in $S$.

**Definition:** The minimum number of elements in a dominating set is called the domination number of the graph and is noted as $\gamma(G)$\(^3\).

- Unrestricted

This is the simplest case, in which there are no restrictions regarding the dominating set. The unrestricted dominating set only consists on the minimal possible set of nodes which dominate the graph.

---

\(^2\)Eventhough only the first three are implemented in GTapp, the algorithms for the rest are proposed in section 4.4.

\(^3\)Note that the proposed algorithms do not guarantee $|D| = \gamma(G)$
2.3. Dominating set

- Connected

This type of domination forces the dominating set to be connected. As defined in section 2.1, a graph or a set of nodes in a graph are connected if there exists a path between every pair of vertex.

**Definition:** If the subgraph generated by \( S \) is connected, then \( S \) is a connected dominant set. The minimum number of elements in a connected dominating set is called the *connected domination number* of the graph and is noted as \( \gamma_C(G) \).

- Independant

Opposed to the previous domination, the independant dominating set establishes that the obtained set must be independant. This means that nodes contained in the dominating set can not be neighbours among them.

**Definition:** If the subgraph generated by \( S \) is independant, then \( S \) is an independant dominant set. The minimum number of elements in an independant dominating set is called the *independant domination number* of the graph and is noted as \( i(G) \).

- Total

The total domination restriction demands the dominating set \( D \) not to contain independant nodes. This means that in \( D \) there can be several connected components, but none of them can be made up of only one node.

**Definition:** If the subgraph generated by \( S \) does not have isolated nodes, then \( S \) is a total dominant set. The minimum number of elements in a total dominating set is called the *total domination number* of the graph and is noted as \( \gamma_t(G) \).

- Paired

This domination restriction requires the resulting dominating set to be made up of any number of connected components, all of which have to consist on two adjacent nodes.

- Distance 2

This kind of dominating set is not obtained upon a restriction but by varying the notion of neighbourhood. The unrestricted distance 2 dominating set is obtained just by considering the neighbours of a vertex \( v \) to be its neighbours’ neighbours: \( \text{neigh}(v) \to \text{neigh}(\text{neigh}(v)) \).

Based on this idea, we can extend the distance 2 notion in two different directions. On one hand we could simply apply the restrictions previously mentioned for dominating sets. On the other hand, we could obtain other dominating sets if, instead of considering distance 2, we consider distance 3, distance 4, and further. This is known as *distance k*: \( \text{neigh}(v) \to \text{neigh}^k(v) \).
Chapter 3

Computer science ground

Besides the mathematical ground we have previously covered, we will now consider the computer science knowledge necessary to understand how the application was structured. More specifically, we will discuss the different programming paradigms involved and their avail.

These paradigms establish a methodology and a style for implementing functionalities, serving as a way for improving the quality of software by enforcing logical structures. Even though there is a large number of different paradigms, in this portfolio we will only address those with a relevant impact on the project. Three main paradigms coexist within GTapp. Object Oriented Programming (OOP) was the paradigm used for structuring code and data while Event-Driven Programming (EDP) detailed how the user interface would react to user input. Together with OOP and EDP, GTapp was implemented under Structured Programming (SP).

3.1 Structured programming

Structured Programming is aimed at decomposing the code structure into smaller modules. Each of these modules will contain a particular functionality which is decoupled from the rest of the program. This praxis allows each of the modules to be tested separately before integrating with other modules [25]. Moreover this architecture avoids common pitfalls of non-structured languages that rely on a heavily disciplined developer. Not only will the code be more readable, characteristic which increases code maintainability, but the organization of the program into separate pieces allow for software reusability and scalability.

Structured Programming is now encouraged by modern procedural languages by providing several control structures to handle the flow of execution. These structures are subroutines, block structures and loops. On one hand subroutines and block structures favor breaking up large sections of code, enabling the execution flow to go on from one unit of work to another. On the other hand loops allow a statement to be executed repeatedly until a termination condition is met. In contrast to non-structured programming, Go-to statements are avoided, being replaced by other structures like nested conditionals and loops. Furthermore, it
has been demonstrated that any computer program can be implemented by using these structures [3][22].

### 3.2 Object oriented programming

Object oriented programming is based on the concept of **objects**. These are units of work which may contain data and/or runnable code. Inside an object, data is stored in the form of fields which are known as **attributes**. Data can be specified to be read-only or writable as well as global or local to the object, among other constraints. Functionalities, in the form of procedures are known as **methods** and represent portions of code which may receive parameters and return a result. Methods can modify the state of values or attributes inside the same object, but OOP is aimed towards object interaction. Objects can interact between them, calling methods inside another object, if allowed, and likewise modifying its attributes.

Object oriented languages are often simultaneously class-based, meaning that objects are instances of classes, which define their type and characteristics. This feature allows for encapsulation, composition and inheritance.

- **Encapsulation** restricts the access to the components of an object, allowing only the desired members of a class to be exposed while being able to hide the implementation details.[7]

- **Composition** implies that an object can contain another object in its attributes. We refer to an example for easier understanding: in GTapp a list of Node objects belong to another object called Canvas.

- One of the most significant concepts in OOP is **Inheritance**, which allows classes with similarities among them to share code. Inheritance provides mechanisms for a class to extend another class, both commonly known as child and parent respectively. Again concerning GTapp, each of our particular dominating set classes (e.g. Conn Dom) inherits from an abstract [1] dominating set class (A_Dom) with common code inherited by each child, same as with triangulation classes.

### 3.3 Event driven programming

Event driven programming is a paradigm based on the concept of signals or **events**, such as user action (e.g. mouse click) or messages from other programs[10]. An event-driven application usually runs a main loop which listens for events and triggers callback functions when activated, which determine the flow of execution throughout the program [12]. Contrary to Procedural Programming and goal based programs, event driven programming provides a reactive behavior. This is essential for Guided User Interfaces (GUI), which rely on user input to execute tasks [11].
A common software solution for implementing event driven programs is the observer pattern\(^1\). This event handling pattern relies on observers which subscribe to subjects. When an event is triggered and received by a particular subject, it broadcasts a notification to all of its observers. This mechanism allows several observers to subscribe to a particular subject meaning that a singular event can trigger different code sections.

A key feature provided by the C\# language for implementing this pattern are delegates [9][13]. The event model uses delegates to bind event handlers (observers) to events (subjects). In other words, event handlers are bound to events so that when an event is triggered, the code within all of its event handlers is executed.

The following section of code (Listing 3.1) belonging to GTapp offers a good example of how event handling works. The extracted code belongs to the class Form1 which stands for the main GTapp window. Furthermore, the event handler shown is the one triggered after this window is being resized. The reader may notice that upon line eleven the event handler function Form1_Resize is bound to the event SizeChanged. The handler implementation can be found between lines fifteen and twenty two, and it consists on obtaining the screen coordinates, creating a string message from them and assigning that message to the bottom label.

\(^1\)Do not misunderstand the observer pattern with the pub-sub pattern, which is related but applies more specifically to messages between elements.
### 3.4 Language and IDE

As we have stated before, for this project C-Sharp was the preferred programming language. C# is a general-purpose, object oriented programming language, alike Java. Even though the background of the student was based on this language, taught in the university for several years, C# was chosen over Java not only as a way of learning a new language but also because of the advanced functionalities it provided\(^2\). For instance event delegates, functions that enable simple event-driven programming, do not exist in Java. Moreover the language equips the developer

\(^2\)We link to an interesting article explaining the differences among Java and C-Sharp. [4]
with a powerful tool that Java still misses after all this time: a dedicated Integrated Development Environment (IDE).

Visual Studio is the IDE built from Microsoft for developing Microsoft applications, not only in C# but also in other Microsoft languages like C, C++ or Visual Basic. Furthermore, Visual Studio supports other languages such as Python or Ruby via separately installed language services. Meanwhile there are plenty of IDEs for Java, both free like Eclipse or Netbeans and commercial like IntelliJ, but Visual Studio provides a unique and compelling engine. Visual Studio’s code editor includes a exclusive technology for code completion and refactoring called IntelliSense as well as an integrated debugger.

Besides the previously mentioned delegate feature not existing in Java, C-sharp and Visual Studio together also grant developers a hard to ignore programming platform: Windows Forms. Windows Forms is an intuitive Guided User Interface (GUI) library which is integrated within Visual Studio. The Microsoft IDE was created with Windows Form support, granting developers a programmable GUI on-the-fly over a graphical previsualization. In the following snapshot (Figure 3.1) we present the Visual Studio environment layout as well as the recently indicated previsualization.

Figure 3.1: Visual Studio IDE snapshot showing code editor and Windows Form preview.
Chapter 4
GTapp

Along chapter 4 we will dig into the implemented application GTapp. Created as a tool for generating graphs, triangu-
ulating and calculating dominating sets, GTapp quickly became the centerpiece of this final project. We will try to break down
the application by standing at different points of view from where we can take a closer look into each of its different modules.

First the folder structure and the relation between units of work will be dis-
cussed, afterwards we shall discover how the execution flows from the application boot. Finally, this section will finish with the mathematical algorithms for trian-
gulations and dominating sets.

4.1 Program Structure

The Visual Studio framework divides solutions into projects and projects can be
divided as well into folders with source files. Figure 4.1 represents the directory
structure of the Visual Studio solution, called TFG. Inside the solution, the project
also named TFG can be found. Further inspection reveals the five main folders
of the application: Configuration, Dominations, Graph, Triangulations and Wind-
dow.

Leaving aside this folders for a moment, inside the project folder other minor
files can be noticed. Properties holds information about the assembly as well as
resources and settings. Inside References, the external libraries used are located.
The folder Resources owns all the icons, except the desktop icon, which can be
found at the bottom of the TFG folder named as Logo4.ico. Last but not least,
App.config is a small file with information about the encoding and framework
version, and Program.cs is the entry point for GTapp.

Now we go through each of the foremost folders one at a time.

1A diagram showing the relation between each component can be found at appendix A.
2Note that C# filename extension is .cs.
4.1.1 Configuration

*Configuration* folder holds the files `Config.cs` and `Messages.cs`.

- `Config.cs` sets up the application’s default values which can be later modified while the application is running, through the settings popup window. Furthermore, it is in the `Config` class where we store two arrays representing the possible triangulation and dominating set algorithms, together with their indexes for external referencing\(^3\).

- `Messages.cs` on the other hand hosts a pack of methods which return string values to be used when refreshing the bottom label in *GTapp*. We decided to take this messages away from the source code into another separate file not only for future developers to clearly identify all the possible output messages, but also looking forward a potential translation of the application, for which the stated methods could return a particular text string depending on the selected language.

\(^3\)Each index univocally corresponds to an algorithm, so that if an external class references a particular algorithm, the config file will be in charge of returning the appropriate algorithm.
4.1.2 Graph

Inside *Graph* we can find the files *Node.cs*, *Edge.cs* and *Canvas.cs* which define the essential elements of a graph.

- **Node.cs** defines a node as a pair of \(x\) and \(y\) coordinates, with a list of nodes as neighbours as well as a couple of booleans to determine if the node is dominant or being dominated\(^4\). This file also specifies some behaviour to a node, like adding a neighbour, mark as dominated, paint onto a canvas or `ToString()` and `Equals()` methods for representing a node as text and compare a couple of nodes respectively, among others.

- **Edge.cs**'s structure is similar to the previous file but instead of defining a node, in this case it defines the edge component. An edge is simply defined as a pair of nodes. The functionalities are shortened to `Paint()`, `Equals()` and `ToString()`.

- **Canvas.cs** is one of the fundamental files within the application. Besides holding the set of nodes and edges available on screen, data structures about dominating nodes, selected vertices and convex hull edges are held in the *Canvas* class simultaneously. In addition it is in this class where the selection functionality\(^5\) is located, together with information about which triangulation and dominating set algorithms are selected at a given moment. Regarding functionalities, *Canvas* is in charge of the logic related to graph handling. Within the range of its capabilities we highlight painting the graph (triangulation and dominating set included), adding or removing nodes and finding a vertex upon its coordinates.

4.1.3 Dominations

Those files in charge of dominating set algorithms can be found in *Dominations*.

- **A_Dom.cs** is an abstract class which is implemented by the other files within the folder. Inside the class we define an abstract method\(^6\) called `Algorithm()`, which will be filled with specific code for each of the implementing members. Besides there are two auxiliary methods, `domination_complete()` for checking if a set of nodes is fully dominated and `greatest_degree()` for acquiring from a list of nodes the one with more neighbours.

- **Minimal_Dom.cs**, **Connected_Dom.cs** and **Indep_Dom.cs** only implement the previously mentioned algorithm for a generic dominating set, a connected and an independent one respectively\(^7\).

- The files **Distance_Dom.cs**, **Paired_Dom.cs** and **Total_Dom.cs** were included in the project with the intention of being completed on a future work on dominating sets.

\(^4\)We reference the following article [19] which inspired this implementation.

\(^5\)By selection functionality we understand dragging an area on screen with middle click for selecting a number of nodes.

\(^6\)See section 3.2 for more details on abstract classes and inheritance.

\(^7\)See section 2.3 for more details on dominating set algorithms.
4.1.4 Triangulations

Same as with Dominations folder, in Triangulations we can find an abstract class A_Triang.cs and all of the members which inherit from it.

- A_Triang.cs is in charge of declaring a method for the algorithm implementation while all the classes which implement this abstract class will have to detail the precise algorithm. As well as A_Dom.cs, this class also provides common auxiliar algorithms, but in this case we can find a larger number of functions. Some examples of these functionalities are: finding the orientation of a node regarding other two vertex, swapping the bottom-most node in a list of node with the first position of the list, calculating the distance between two nodes, ordering a list by polar angle as well as deleting duplicates, and checking if a couple of edges overlap.

4.1.5 Window

Window folder contains all the different windows that can be seen in the application.

- mainForm.cs (Figure 4.2) is the starting and leading window in GTapp, every other window spawns from this one. mainForm can be visually divided into 4 elements.
  From the bottom, the first available element is the Bottom-Label, where the state of the application is displayed. This label is refreshed whenever the user performs an action and gives back more particular information about the performed action (e.g. number of nodes added or coordinates of selected vertex). In the middle of the screen we can find the canvas, where graphs are drawn. By clicking over this element the user can add nodes (left click),

![Main window, screenshot taken upon boot.](image-url)
remove a particular node (right click) or select nodes positioned on a region (middle click). Above there is a toolstrip with shortcut icons for common behaviours like clearing the canvas or adding random nodes, and on the upper part of the window there is a menustrip with the typical menus: File, Edit, View and Help.

![Configuration window screenshot.](image)

Figure 4.3: Configuration window screenshot.

- **ConfigForm.cs** (Figure 4.3) was designed in order to be able to customize certain parameters concerning graph visualization. Within this screen a user can change the size of nodes and edges as well as the color of nodes and edges according to their nature. Two sliders where used for size modification, each of which restricted the maximum and minimum values to prevent unexpected or unreasonable values. In addition, a tooltip is shown when hovering over the node size slider.\(^8\)

Due to the fact that in **GTapp** colours are represented thanks to the **.Net Framework Color structure** [5], the **colourPicker** dialog could be use for intuitive colour selection. This dialogs are shown after performing a click over an specific colour selection button. This popup windows are frequently used in Microsoft Windows applications and the user is meant to be familiar with this kind of dialogs.

---

\(^8\)This tooltip warns the user about the possibility of unexpected behaviour when increasing the size of nodes after two or more have been placed close together in the canvas. This behaviour is unavoidable because nodes happen to overlap.
• **AutomaticExecutionForm.cs** (Figure 4.4) offers the possibility to perform an automatic execution and save the obtained data into a spreadsheet file. In this window the user can set up the number of nodes and number of iterations (or number of times the execution is going to be run over). After selected, a prompt dialog will be shown for the user to choose the name and location of the spreadsheet file in which the gathered results will be stored. This data is formatted to fit into the custom excel file presented in section 5.

Figure 4.4: Automatic execution window screenshot.

• **RandomNodesForm.cs** holds the implementation of the Random nodes window (Figure 4.5), which may be triggered from two different buttons. Accessible from *Add Random Nodes* and *Remove Random Nodes* inside the *Edit* menu, this dialog window is meant to allow the user set the number of random nodes desired to add or remove respectively.

Figure 4.5: Random nodes window screenshot.
4.1. Program Structure

Figure 4.6: Information window screenshot.

- **InfoForm.cs** (Figure 4.6) displays a set of parameters calculated from the previewed graph. Thanks to this window, the information about the number of nodes, the number of edges, the number of nodes in the convex hull and the number in the dominating set is automatically calculated, without the need to count manually.

Figure 4.7: About window screenshot.

- **AboutForm.cs** (Figure 4.7) is a read-only dialog window where the copyright information can be seen. Besides the logo and title of the application, also the author, the author’s website, the year and the university references are printed in this window.
• **UserManualForm.cs** (Figure 4.8) serves as a guided tour for GTapp. This window contains different tabs with the necessary knowledge to use the application. The information provided is divided into six sections. First *Canvas* explains how to interact with the canvas. Later four tabs explain the functionalities arranged in the *mainForm* menustrip (figure 4.2). Lastly there is an additional tab to explain how the automatic execution functionality works.

![User Manual](image)

Figure 4.8: User manual window screenshot in Canvas tab.

• **Loading.cs** (Figure 4.9) implements a simple dialog shown while the automatic execution algorithm is being performed. When the execution starts running, this window will prevent any further interaction with the application until the calculations are finished. The window is in charge of displaying the text "*Loading Automatic Execution*" in the Segoe Print font family. We used this font in order to make the window more user-friendly and good looking, due to the fact that meanwhile the application is calculating, the user will only be able to see this dialog.

![Loading](image)

Figure 4.9: Loading window screenshot.
4.2 Painting execution flow

Even though GTapp is oriented towards executing appropriate code depending on captured user interaction (events), and the execution flow directly depends on the user input, there is a particular occasion when the execution goes through many members of the program.

One of the first big challenges faced in this project was the canvas and how to paint graphs dynamically. After implementing the data structure for both nodes and edges, being able to print them on screen was not an easy task. Onwards we explain how the execution flows from when the user clicks on the canvas until when the graph is printed.\footnote{Code fragments have been shorten for an easier understanding.}

For the program to start the execution, assume an user left clicks on the canvas in the first place. This event is captured by the \textit{mainForm} class. The following fragment shows the piece of code where this event is handled. The reader may notice that a new node is added by calling the canvas method \texttt{addVertex} in line ten. If this method does not fail, then the bottom label is updated with the proper message and later the canvas is refreshed.

```csharp
private void on_MouseDown(object sender, MouseEventArgs e)
{
    // Depending on the pressed button
    switch (e.Button)
    {
    // If left click
    case MouseButtons.Left:
        // Add vertex
        if (this.canvas.addVertex(new Node(e.X, e.Y)))
        {
            this.toolStripStatusLabel1.Text =
                this.messages.on_node_insertion(e.X, e.Y, canvas.number_nodes());
            this.paint_canvas();
        }
        break;

    // If right click
    case MouseButtons.Right:
        (...)
        break;

    // If middle click
    case MouseButtons.Middle:
        (...)
    }
```

Listing 4.1: Fragment of code handling the click event over the canvas.

The code for refreshing the screen is shown afterwards. This code is a private (auxiliar) function belonging to the same class `mainForm`. Here we call the method `paint` inside `Canvas`.

Listing 4.2: Fragment of code calling the paint method in Canvas.

Now the `Canvas paint` method comes into play. This function is in charge of calling the rest of the members participating in the graph calculation. First of all the previous calculated data is removed, so that there are no conflicts with current execution. After creating a graphics object in which the program is able to draw the triangulation will be painted (if there is any selected) and the vertex on top of the edges. [15]
if (vertices.Count > 1 && selected_triangulation != null)
{
    // Paints edges (triangulation)
    status += this.paint_Triang(form, formGraphics);
}

if (vertices.Count > 0)
{
    // Paints nodes
    status += this.paint_Nodes(form, formGraphics);
}

    // Garbage collector
formGraphics.Dispose();

    status += "}
return status;

Listing 4.3: Fragment of code in Canvas in charge of painting the graph.

In order to paint the triangulation and the nodes, two other auxiliar methods are invoked. First, `paint_Triang()` calls the method `Algorithm()` belonging to the `selected_triangulation` attribute\(^\text{10}\) and afterwards loops over each of the retrieved edges in order to call the `paint()` method in the `Edge` objects and set the node adjacency.

\texttt{private string paint_Triang(TFG.Form1 form, System.Drawing.Graphics formGraphics)
{
    // Delete the list of neighbours in each node
    foreach (Node node in vertices)
    {
        node.remove_neighs();
    }

    // Variable initialization
    List<Edge> new_edges = new List<Edge>();
    List<Node> convexHull_vertex = new List<Node>();

    // Obtain the triangulation edges
    selected_triangulation.Algorithm(vertices, ref new_edges, ref convexHull_vertex);

    // If the triangulation has any edges
}\textsuperscript{10}This attribute will be an object inheriting from `A_Dom`, which contains the algorithm for calculating the selected triangulation.
if (new_edges != null && new_edges.Count > 0)
{
    foreach (Edge edge in new_edges)
    {
        edge.paint(formGraphics);

        // Add adjacency
        edge.a.add_neigh(edge.b);
        edge.b.add_neigh(edge.a);
    }
}

return " triangulation,";

Listing 4.4: Fragment of code painting triangulation.

The following code is the one related to the node painting. In this fragment we notice that in case there is a triangulation selected different to convex hull and a dominating set selected as well, the algorithm is performed and later the paint method in each of the nodes of the graph is executed.

private string paint_Nodes(TFG.Form1 form,
{
    // Returned string
    string ret = "";

    // If there is both a valid triangulation and a domination selected
    if (selected_triangulation != null &&
               !(selected_triangulation is Convex_Hull) &&
               selected_domination != null)
    {
        // At the beginning there are no dominant or dominated nodes
        foreach (Node node in vertices)
        {
            node.dominate = false;
            node.is_dominated = false;
        }

        List<Node> dom_vertex = new List<Node>();

        // Calculates dominating nodes
        selected_domination.Algorithm(vertices, ref dom_vertex);

        this.dominatingSet_nodes = dom_vertex;
4.3. Triangulation algorithms

For simplicity we have not included the code for the respective Edge and Node painting algorithms. We encourage the reader to study the files Edge.cs and Node.cs for further information about how the painting is performed. Furthermore, we provide additional resources that were helpful to understand some key concepts in output painting. [16] [17][2]

To sum up, the purpose of this section was to give an example on how the different classes coexist within GTapp. In this particular case we have seen how after an user clicks on the canvas, the execution flow goes from mainForm to Canvas, Node, Edge and other auxiliary functions inside those classes, until the graph is fully calculated and printed.

### 4.3 Triangulation algorithms

After solving how to print content into the screen, the project’s biggest challenge emerged: implementing triangulation algorithms. Despite the fact that most triangulation algorithms have already been implemented in other languages, due to the nature of our implementation we had to build a custom algorithm for it to work correctly. Previously implemented algorithms not only were they implemented in languages older than C#, but they also did not guarantee a one hundred percent
rate of success. Due to the fact that our algorithms were going to be executed automatically and with a large amount of nodes, we needed the procedures to be accurate. Most of the problems in algorithm precision came to be collinear points\textsuperscript{11}. Cases in which nodes happen to be collinear are rare but tend to take place more times the more nodes the graph has. Therefore we implemented two triangulation algorithms based on the custom convex hull algorithm. Note that the source code for these three algorithms is available at the appendix B.

4.3.1 Convex hull

Considered as a triangulation algorithm because every other triangulation algorithm is based on this one, the convex hull was the first functionality to be implemented. Our algorithm is based on the Graham Scan algorithm \cite{24,6}, but as stated in the introduction of this section, it has been customized to behave properly under a high amount of nodes \cite{14} and under collinear circumstances. We describe the algorithm ahead.

Let candidate\_vertices be the list of nodes in the graph $G$, and $S$ the stack of nodes in the convex hull.

- Begin.
- If there are at least three nodes in the graph, continue.
- Place the bottommost node $N_0$ in $G$ at the first position of candidate\_vertices.
- Calculate the polar angle of every vertex in $G$ regarding $N_0$.
- Consider the polar angle of $N_0$ to be infinite.
- Order de nodes in candidate\_vertices according to their polar angle, and remove duplicates.
- If there are at least three nodes in candidate\_vertices, continue.
- Push the first three nodes of candidate\_vertices into $S$.
- For each node $i$ in candidate\_vertices:
  - While the first two elements of $S$ and $i$ orientation is positive\textsuperscript{12}, remove the first element of $S$.
  - Push $i$ into $S$.
- End.

\textsuperscript{11}Collinear points are known to be nodes located within a common line.
\textsuperscript{12}The orientation of a node regarding two other nodes is calculated by the determinant of their coordinates.
4.3. Layered triangulation

The algorithm for calculating the layered triangulation is strongly related to the convex hull algorithm. The reason for this bonding is because the layered triangulation consists on calculating consecutive convex hulls and triangulating the space in between. The pseudocode for calculating this particular type of triangulation is the following.

Let \textit{convex\_hulls} be the list with consecutive convex hulls, \textit{outer\_hull} the convex hull of the graph \( G \), \textit{inner\_hull} the last convex hull, and \textit{edges} the edges belonging to the layered triangulation.

- Begin.
- If there is at least one node in \( G \), continue.
- Let \( G' \) be \( G \).
- Do:
  - Add \( CHG' \), the convex hull of \( G' \), to \textit{convex\_hulls}.
  - Remove \( CHG' \) from \( G' \).
- Until \( CHG' \) has less than three nodes.
- For \( i \) in \textit{convex\_hulls}:
  - For \( j \) in \textit{convex\_hulls}[i−1]:
    - For \( k \) in \textit{convex\_hulls}[i]:
      - Insert the Edge \( e \) from \textit{convex\_hulls}[i−1][j] to \textit{convex\_hulls}[i][k] into \textit{edges} if it does not intersect with any in \textit{edges}.
- For \( i \) in \textit{inner\_hull}:
  - for \( j = i + 2 \) in \textit{inner\_hull}:
    - Insert the Edge \( e \) from \textit{inner\_hull}[i] to \textit{inner\_hull}[j] into \textit{edges} if it does not intersect with any in \textit{edges}.
- End.

4.3.3 Random triangulation

Not as related as the layered triangulation algorithm, the random triangulation code also depends on the convex hull. More importantly, the algorithm described in this section is based on creating random edges among nodes that do not overlap, besides the outer hull always included in every possible triangulation of nodes. We refer to the pseudocode for calculating random triangulations in the following lines.
Let \textit{convex hull} be the list of nodes in the convex hull and \( e \) the number of edges in the graph \( G \).

- Begin.
- If there are at least three nodes in \( G \), continue.
- While \( e \neq 3|G| - |\textit{convex hull}| - 3 \) \(^{13}\):
  - Calculate the first random number \( \textit{rnd}_0 \), between 0 and \( |G| - 1 \).
  - Do:
    - Calculate the second random number \( \textit{rnd}_1 \), between 0 and \( |G| - 1 \).
    - Until \( \textit{rnd}_1 \neq \textit{rnd}_0 \) and non-collinear.
    - Insert the Edge \( e \) from \( \textit{convex hull}[\textit{rnd}_0] \) to \( \textit{convex hull}[\textit{rnd}_1] \) into \( \text{edges} \) if it does not intersect with any in \( \text{edges} \).
- End.

### 4.4 Dominating set algorithms

Unlike with triangulation algorithms, dominating set algorithms’ scope is only reduced to node adjacency. This fact is what makes them easier to understand than triangulations. But we should not be fooled by their apparent simplicity because inside lies a problem that does not exist in graph triangulations. As we pointed out in chapter 2.3, we can restrict the dominating set to certain constraints, depending on which we obtain different sets. Due to the nature of this rules, in order to obtain the dominating set with minimal cardinal, we should test all the possible dominating sets of a graph. Eventhough in this project we have proposed several algorithms for different restrictions on dominating sets, they do not necessarily provide the best dominating set possible under those conditions. However, the algorithm is likely to grant a dominating set whose cardinal is acceptably close to the optimal in a randomly distributed set of nodes \(^{14}\).

We will present a simplified version of the algorithms, raised as part of the project, in the following subsections. The complete code for the implemented algorithms is available at the appendix B.

#### 4.4.1 Greedy algorithm

The greedy algorithm calculates the dominating set of a graph without any restrictions.

\(^{13}\)Equation obtained from \( e = 2V_B + 3V_I - 3 \). \cite{20}

\(^{14}\)There are cases in which our algorithms are far from providing an optimal set, but this cases are known to be singular and unlikely to happen.
Let \( D \) be the dominating set of \( G \) and \( n \) the node with the greater number neighbours in \( G' \) during each iteration.

- Begin.

- Do:
  
  - Consider \( n \) to be the node with more neighbours in \( G' \).
  
  - Add \( n \) to \( D \).
  
  - Remove \( n \) and all of its neighbours from \( G' \).

- Until every node \( n \) in \( G \) is dominated by a node in \( D \).

- End.

This simple method is called greedy because it only looks for nodes with the greatest amount of neighbours. Although this algorithm may work for randomly distributed graphs like the one in figure 4.10, it is known to perform badly on particular graphs like in figure 4.11.

![Figure 4.10: Example of an optimal execution of the greedy algorithm.](image)

(a) Best unrestricted dominating set. (b) Greedy algorithm dominating set.

![Figure 4.11: Example where the greedy algorithm does not obtain the optimal dominating set.](image)
4.4.2 Connected algorithm

In this particular case we have added a condition to the greedy dominating set algorithm. This condition states that the dominating set must be connected together. So, even though the algorithm will still pick the nodes with more neighbours, now these nodes are taken from the set of neighbours of previously selected dominating vertex.

Let $D$ be the dominating set of $G$ and $n$ the node with the greater number of neighbours in $D$ during each iteration.

- Begin.
- Let $n$ to be the node with more neighbours in $G$.
- Add $n$ to $D$.
- While there is at least one node in $G$ not being dominated:
  - Consider $n$ to be the node in $\text{neighbours}(D)$ with more neighbours in $G - D$.
  - Add $n$ to $D$.
  - Add every neighbour of $n$ to $D$.
- Until every node $n$ in $G$ is dominated by a node in $D$.
- End.

Equivalent to how we demonstrated with the greedy algorithm, the connected algorithm does not obtain the dominating set with the less number of nodes for every possible graph. The following figure 4.12 shows a counterexample which proofs the previous statement.

![Figure 4.12: Example where the connected algorithm does not obtain the minimal dominating set.](image)

4.4.3 Independant algorithm

Regarding the independant dominating set, the restriction applied is that any two nodes in the dominating set cannot be neighbours. In this case the algorithm still takes the nodes with greater number of neighbours, but only those that are not
Let $D$ be the dominating set of $G$ and $n$ the node with the greatest number
neighbours in $G - (D + \text{neighbours}(D))$ during each iteration.

- Begin.
- Do:
  - Consider $n$ to be the node in $G - (D + \text{neighbours}(D))$ with more
    neighbours in $G - D$.
  - Add $n$ to $D$.
- Until every node $n$ in $G$ is dominated by a node in $D$.
- End.

Probably the reader has already inferred that this algorithm behaves exactly
like the previous two regarding calculating the minimal dominating set. We ad-
dress figure 4.13 which proofs that the independant algorithm does not necessarily
provide the dominating set with less vertex in every possible graph triangulation.

![Figure 4.13](image)

(a) Best independant dominating set.  (b) Independant algorithm dominating set.

Figure 4.13: Example where the independant algorithm does not obtain the opti-
mal dominating set.

### 4.4.4 Total algorithm

The algorithm for calculating the total dominating set $D$ of a graph $G$ is proposed
to be the following.

- Begin.
- Do:
  - Consider $n$ to be the node with more neighbours in $G - D$.
  - Add $n$ to $D$.
- Until every node $n$ in $G$ is dominated by a node in $D$.
- For every node $n$ in $D$ which is independant:
  - Add an edge between $n$ and a node in $D$
• End.

We point out that this algorithm may result in the same dominating set as the connected algorithm if the nodes with more neighbours happen to be connected.

Our first approach to this algorithm was to begin by choosing the node with the highest degree in $G$ and afterwards the neighbour with higher degree $G - D$, to later under a certain probability, choose whether to pick a neighbour of $D$ or an independent node. Although it may be interesting to study the degree of $D$ regarding different probabilities, this algorithm does not necessarily provide a better dominating set than the previous algorithms in terms of number of vertex.

### 4.4.5 Paired algorithm

The algorithm for calculating the dominating set $D$ under the paired restriction for a graph $G$ is recommended to be the following.

• Begin.

• Do:
  
  – Consider $n_0$ and $n_1$ neighbours with higher combined degree\(^{15}\) in $G - D$.
  
  – Add $n_0$ and $n_1$ to $D$.

• Until every node $n$ in $G$ is dominated by a node in $D$.

• End.

### 4.4.6 Distance 2 algorithm

This algorithm does not constrain the dominating set to a restriction, as with the greedy algorithm, but in this case we consider adjacency to apply two steps at a time. In other words, the neighbours of a node will be the neighbours of its neighbours. By pushing the vicinity definition a step further we may obtain new and smaller dominating sets for the same graphs. Therefore, due to the fact that only the neighborhood has changed and not the restrictions, the algorithms for calculating dominating sets under constraints remain the same. Likewise this applies for the generalization of this consideration on dominating sets: distance $k$.

\(^{15}\)By combined degree we understand $\deg(n_0) + \deg(n_1) - \deg(n_0)\ |

\n\n\deg(n_1)$, where $n_0$ is the degree of those nodes adjacent to both $n_0$ and $n_1$. 


Chapter 5

Experimental research

When the implementation of GTapp concluded, we started working on the experimental results. Like we previously mentioned back in section 1.2, the final purpose of this application was not only to be able to visualize graphs, but to obtain experimental data about the behaviour of dominating sets within triangulations. With this in mind, the last piece of code produced was the automatic execution functionality. This led to a soft transition between the computer science side of the project towards the mathematical.

For the statistical analysis we used the Microsoft spreadsheet program Excel. Excel forms part of Microsoft Office, being the industry standard for spreadsheets. Moreover, by using Microsoft Visual Studio, data saving was done by the use of a library. The code used to save data into an excel file can be seen at the appendix D.1.

5.1 Storing data

After selecting the desired number of nodes and iterations in the Automatic Execution window (figure 4.4), GTapp stores the captured data into the beforehand specified .xls file\footnote{Note that spreadsheet filename extension is .xls.}. This file (see figure 5.1) contains the number of nodes in the dominating set, the number of nodes in the convex hull and the number of edges in the triangulation for each and every possible combination of implemented algorithms\footnote{A total of six rows for the possible combinations between Layered and Random triangulation matched against Connected, Independant and Greedy dominating sets. For each of these six combinations, three statistics were obtained.}. The details on how this process is executed can be found in appendix D. Keep in mind that the generated file holds a gap between the meta-data and the values retrieved. The purpose of this gap is to be later filled with statistical measures [21].
In order to produce diagrams on the obtained data, we had to store it inside a unique excel file. In the file `graficas.xls` we pasted the results obtained after thirty executions of the automatic algorithm. Arranged by triangulations and dominating sets, we calculated the average, the mode, the variance and the minimum and maximum for the 24,630 captured values. Due to the time the program took to compute the algorithms, the samples were irregular: 250 samples were taken for graphs with between 25 and 100 nodes, and 180, 100, 50 and 35 samples for 125, 150, 175 and 200 nodes respectively. Figure 5.2 shows a fraction of the whole file. Notice that the data was distributed along 193 rows and 260 columns and the picture only provides 40 rows and 60 columns.
5.2 Data mapping

Obviously, just by having a look at the data in the table it was impossible to obtain any knowledge, so we decided to build graphs based on this statistics. Therefore we created three different graphs comparing the number of vertex in the graph $V$, the number of nodes in the convex hull $K$ and the number of nodes in a particular dominating set $D$.

5.2.1 D/V Graph

This first graph (figure 5.3) shows the relation between the nodes in the dominating set in the y-axis and the nodes in the graph in the x-axis. We can extract from the picture two interesting facts. First of all that the relation seems to follow a linear regression as a one degree polynomial$^3$. Secondly, we can estimate how good or bad are the dominating sets found by these algorithms. On average, the best dominating set algorithm has been found to be the independant, while the worst algorithm is the connected dominating set for random triangulations. In between, both greedy dominating sets and the connected for layered triangulations are not distant from each other, with roughly a difference of four nodes when the graph has 200 points.

Moreover, if we compare the relation between $D$ and $V$ to the relation between $K$ and $V$ we find out that graphs with more than 50 vertex will most likely have more nodes in their dominating set than in their convex hull. Likely we point out that for graphs with 25 nodes, there will be more nodes in the outer hull than in the dominating set.

\[\text{Figure 5.3: Graph showing } D \text{ over } V\]

$^3$The equation for the polynomials is given afterwards, in figure 5.4.
Besides the previous information obtained by looking at the graph, we could calculate the linear regression of the data for each particular algorithm. In appendix D.2.1 we address the different graphs plus their estimated polynomial. In the following charts we bring together the equations of these polynomials (figure 5.4), plus the experimental boundaries (figure 5.5) obtained for the number of nodes in D depending on each particular algorithm.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Greedy</th>
<th>Connected</th>
<th>Independant</th>
</tr>
</thead>
<tbody>
<tr>
<td>Layered triangulation</td>
<td>$y = 0.2732x - 1,5105$</td>
<td>$y = 0.2844x - 1,5649$</td>
<td>$y = 0.1997x + 0.553$</td>
</tr>
<tr>
<td>Random triangulation</td>
<td>$y = 0.2907x - 1,5814$</td>
<td>$y = 0.3705x - 3,3907$</td>
<td>$y = 0.1817x + 1,3667$</td>
</tr>
</tbody>
</table>

Figure 5.4: Equations of the regression polynomials of degree one for D/V.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Greedy</th>
<th>Connected</th>
<th>Independant</th>
</tr>
</thead>
<tbody>
<tr>
<td>Layered triangulation</td>
<td>1/3.660</td>
<td>1/3.516</td>
<td>1/5.007</td>
</tr>
<tr>
<td>Random triangulation</td>
<td>1/3.440</td>
<td>1/2.700</td>
<td>1/5.504</td>
</tr>
</tbody>
</table>

Figure 5.5: Experimental boundaries obtained for D/V

5.2.2 D/K Graph

In this case we compare the number of nodes in the dominating set regarding the nodes in the convex hull. Identical to what happened in the previous graph, figure 5.6 provides the same evidence about which algorithm produces more nodes in the dominating set. The greedy algorithm for layered triangulation remains to be the worst possible algorithm while the independant dominating set grant the best results.
Beyond the preliminary analysis, this diagram also shows that the relation does not fit into a degree one polynomial. Now, the relation among $D$ and $K$ follows a regression polynomial of degree two\(^4\). As for $D/V$, the reader may find in the appendix D.2.2 each particular graph plotted together with its polynomial regression. The following chart (figure 5.7) shows the equations of this estimated functions.

\begin{table}[h]
\centering
\begin{tabular}{l|l}
\textbf{Greedy} & \\
\hline
Layered & $y = 1,3341x^2 - 22,236x + 98,782$ \\
Random & $y = 1,05x^2 - 15,925x + 66,055$ \\
\hline
\textbf{Connected} & \\
Layered & $y = 1,486x^2 - 24,762x + 109,84$ \\
Random & $y = 1,7366x^2 - 27,933x + 119,36$ \\
\hline
\textbf{Independant} & \\
Layered & $y = 0,5132x^2 - 5,7215x + 16,483$ \\
Random & $y = 0,8872x^2 - 13,996x + 60,473$ \\
\end{tabular}
\caption{Equations of the regression polynomials of degree two for $D/K$}
\end{table}

\(^4\)Indeed, this polynomial was expected to be of a greater degree than the one estimating $D/V$. 

\[\text{Figure 5.6: Graph showing K over V}\]
Chapter 6

Conclusion

The present portfolio is thought to be an introduction to the main concepts of dominations and triangulations. For that, a GUI application was created in order to study the main ideas in depth. Yet, this paper does not deliver all the conclusions and achievements we can obtain from dominations. This initial approach should serve, not only as a way for increasing the awareness on this properties, but also for guiding any possible future work on dominating sets.

GTapp was made with the intention to generate random data, apply dominating set algorithms, and obtain statistics that we could later use for establishing experimental quotes. As we have seen in section 5, this process was accomplished successfully. Still, there are many more results yet to be discovered derived from dominating sets in triangulations. Here is where the secondary objective of the application comes into play. GTapp was designed as a scalable system from the beginning. Both the software architecture and the meticulous documentation provided allow an easy understanding of the code and effortless inclusion of new capacities.

From the educational point of view, the current project was equally challenging and rewarding. For instance, building a GUI application from scratch was an eager objective which induced to troublesome obstacles, but at the same time it contributed to learning a new way of developing software. Furthermore, encouraged by the choice of a relatively new programming language, all the algorithms had to be thought over and reimplemented within this framework, requiring for a more clear understanding of both the technology and the mathematical algorithms. Elaborating this project was a unique formative experience that perfectly puts an end to the mathematical and computer science degree in which this portfolio is enclosed.

All of the previous lines sum up into a certain recommendation for future college students to pick up this same topic and carry on with the theoretical and experimental results. We want to firmly encourage those mathematics students with a computer science background to use and enhance the application developed in this portfolio, so that in a future GTapp could be a tool used in universities to both teach and research on graph theory.
Chapter 7

Bibliography


Appendices
Appendix A

GTapp file architecture

Figure A.1: Simplified structure and relation between folders in GTapp.
Figure A.2: Detailed structure and relation between components in GTapp.
Appendix B

Triangulations Source Code

B.1 Convex Hull Algorithm

```csharp
using System;
using System.Collections.Generic;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using TFG.Configuration;
using TFG.Graph;

namespace TFG.Triangulations
{
    class Convex_Hull : A_Triang
    {
        public override void Algorithm(List<Node> vertices,
            ref List<Edge> convex_hull_edges, ref List<Node> convex_hull_vertex)
        {
            // Variable auxiliar con la cantidad de vértices
            int num_vertices = vertices.Count();

            // Pila para ir añadiendo vértices del cierre convexo
            Stack<Node> convex_hull = new Stack<Node>();

            // Si hay menos de 3 vértices no hay cierre convexo
            if (num_vertices >= 3)
            {
                // Añadimos todos los vértices en una lista auxiliar
                List<Node> candidate_vertices = new List<Node>();
                foreach (Node node in vertices)
```
B.1. Convex Hull Algorithm

```csharp
    // candidate_vertices.Add(node);

    // Swap de la primera posición con aquella de más abajo (y más a la izquierda)
    swap_bottommost_to_first_position(ref candidate_vertices);

    // Array auxiliar con los ángulos polares
    double[] polar_angles = new double[num_vertices];

    // Consideramos infinito el ángulo del primer nodo consigo mismo
    polar_angles[0] = double.PositiveInfinity;

    // Inicializamos el resto de ángulos polares
    for (int i = 1; i < polar_angles.Length; i++)
    {
        polar_angles[i] =
            calculate_polar_angle(candidate_vertices[0],
                                  candidate_vertices[i]);
    }

    // Ordena vértices por orden de ángulo polar a partir del primer nodo (previamente ordenado como inferior y más a la izquierda)
    candidate_vertices =
        order_by_polar_angle(candidate_vertices, ref polar_angles);

    // Elimina vértices con el mismo ángulo polar (porque no van a estar en el cierre convexo dos vértices con el mismo ángulo)
    candidate_vertices =
        remove_duplicates_with_same_angle(candidate_vertices, ref polar_angles);

    // Si todavía quedan más de 3 vértices candidatos a estar en el cierre convexo
    if (candidate_vertices.Count >= 3)
    {
        // Vamos añadiendo los nodos en una pila, inicialmente los 3 primeros
        convex_hull.Push(candidate_vertices[0]);
        convex_hull.Push(candidate_vertices[1]);
        convex_hull.Push(candidate_vertices[2]);
    }
```
// Para cada uno de los vértices candidatos a estar en el cierre convexo
for (int i = 3; i < candidate_vertices.Count; i++)
{
    // Si el giro es en sentido horario se descarta
    while (orientacion(convex_hull.ElementAt(1), convex_hull.ElementAt(0), candidate_vertices[i]) > 0)
    {
        convex_hull.Pop();
    }
    // Cuando el giro sea a izquierdas, se añade
    convex_hull.Push(candidate_vertices[i]);
}
else
{
    // Si quedan dos vértices, los añadimos al cierre convexo
    if (candidate_vertices.Count == 2)
    {
        convex_hull.Push(candidate_vertices[0]);
        convex_hull.Push(candidate_vertices[1]);
    }
    // Si solo queda un vértice en el lienzo después de borrar duplicados, lo añadimos al cierre convexo
    else if (num_vertices == 1)
    {
        convex_hull.Push(vertices[0]);
    }
}
else
{
    // Si hay dos vértices, los añadimos al cierre convexo
    if (num_vertices == 2)
    {
        convex_hull.Push(vertices[0]);
        convex_hull.Push(vertices[1]);
    }
    // Solo hay un vértice en el lienzo, lo añadimos al cierre convexo
    else if (num_vertices == 1)
Listing B.1: Source code for the Convex Hull algorithm.
B.2 Layered Triangulation Algorithm

```csharp
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using TFG.Configuration;
using TFG.Graph;

namespace TFG.Triangulations
{
    class Layer_triang : A_Triang
    {
        public override void Algorithm(List<Node> vertices,
            ref List<Edge> layer_tri_edges, ref List<Node>
            convexHull_vertex)
        {
            // Si hay vértices en el lienzo
            if (vertices.Count > 0)
            {
                // Duplicamos en una lista auxiliar sobre la
                // que eliminaremos los vertices exteriores
                List<Node> vertices_aux = new List<Node>();
                foreach (Node node in vertices)
                {
                    vertices_aux.Add(node);
                }

                // Inicializamos objeto convex_null que
                // contiene el algoritmo
                Convex_Hull convex_hull_calculator = new
                Convex_Hull();

                // Inicializamos la lista de listas de
                // vértices que contendrán los sucesivos
                // cierres convexos
                List<List<Node>> convex_hulls = new
                List<List<Node>>();

                // Inicializamos la variable de cierre
                // convexo exterior
                List<Node> outer_hull_vertex = new
                List<Node>();

                // Hacemos el cierre convexo y si hay
                // vértices interiores, seguimos
                do
                {
```

B.2. Layered Triangulation Algorithm

```csharp
// Calculamos cierre convexo
convex_hull_calculator.Algorithm(vertices_aux,
    ref layer_tri_edges, ref
    outer_hull_vertex);

// Si hay vértices en el cierre convexo,
// añadimos el cierre a la lista de
// listas con todos los cierres convexos
if (outer_hull_vertex.Count > 0)
{
    convex_hulls.Add(outer_hull_vertex);
}

// Eliminamos vertices exteriores para
// seguir haciendo cierre convexo de lo
// que quede
foreach (Node node in outer_hull_vertex)
{
    findAndRemove(node.x, node.y,
        vertices_aux);
}
} while (outer_hull_vertex.Count >= 3);

convexHull_vertex = convex_hulls[0];

// Inicializamos variable para arista
candidata
Edge candidate_edge = null;

// Para cada uno de los cierres convexos
for (int i = 1; i < convex_hulls.Count; i++)
{
    // Para cada uno de los nodos del cierre
    // convexo superior
    for (int j = 0; j < convex_hulls[i - 1].Count; j++)
    {
        // Para cada uno de los nodos del
        // cierre convexo interior
        for (int k = 0; k < convex_hulls[i].Count; k++)
        {
            // Inicializamos arista candidata
candidate_edge = new
            Edge(convex_hulls[i - 1][j],
                convex_hulls[i][k],
                Config.color_Edge_default);

            // Se inserta dicha arista si no
            hay intersección con otra
```
Insertando las aristas que no interfieren

```
71 insert_edge_if_not_intersect(candidate_edge, layer_tri_edges);

72 }
73 }
74 }
75 // Para cada vértice del último de los cierres convexos (interior)
76 for (int i = 0; i < convex_hulls.Last().Count; i++)
77 {
78 // Para cada vértice [+2] del mismo array (cierre convexo interior)
79 for (int j = i + 2; j < convex_hulls.Last().Count; j++)
80 {
81 // Unimos los vértices interiores saltando de dos en dos
82 candidate_edge = new Edge(convex_hulls.Last()[i], convex_hulls.Last()[j], Config.color_Edge_default);
83
84 // Se inserta arista si no hay intersección con otra arista de la triangulación
85 insert_edge_if_not_intersect(candidate_edge, layer_tri_edges);
86 }
87 }
88 // Damos la vuelta a la lista de aristas para que imprima con prioridad (por encima) el cierre convexo
89 layer_tri_edges.Reverse();
90 }
91 }
92 }
93 }
```

Listing B.2: Source code for the Layered triangulation algorithm.
B.3 Random Triangulation Algorithm

```csharp
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using TFG.Configuration;
using TFG.Graph;

namespace TFG.Triangulations
{
    class Random_triang : A_Triang
    {
        public override void Algorithm(List<Node> vertices ,
                                      ref List<Edge> random_tri_edges ,
                                      ref List<Node> ConvexHull_vertex)
        {
            // Si hay vértices en el lienzo
            if (vertices.Count > 0)
            {
                // Inicializamos objeto convex_hull que contiene el algoritmo del cierre convexo
                Convex_Hull convex_hull_calculator = new Convex_Hull();

                // Hacemos el cierre convexo
                List<Node> outer_hull_vertex = new List<Node>();
                convex_hull_calculator.Algorithm(vertices ,
                                                  ref random_tri_edges ,
                                                  ref outer_hull_vertex);

                ConvexHull_vertex = outer_hull_vertex;

                // Si hay más de tres vértices en el lienzo
                if (vertices.Count > 3)
                {
                    // Inicializamos el generador de numeros
                    // Con semilla igual a la coord x del vértice inferior (y más a la izquierda)
                    Random random_number_generator = new Random(vertices[find_bottommost_node_index(vertices)]);

                    // Variables auxiliares para el índice de los vértices a unir en la lista de vértices del lienzo
                    int rnd_index_from = 0;
                }
            }
        }
    }
}
```
int rnd_index_to = 0;

// Variables auxiliares para la conversión de los índices anteriores en sus nodos
Node from = vertices[0];
Node to = vertices[0];

// Arista candidata a ser insertada
Edge candidate_edge = null;

// Mientras no se complete de aristas la triangulación (fórmula de euler)
while (random_tri_edges.Count != (3 * vertices.Count - outer_hull_vertex.Count - 3))
{
    // Generamos un índice aleatorio
    rnd_index_from =
    random_number_generator.Next(0, vertices.Count);

    // Generamos números aleatorios hasta que sean distintos 'from' y 'to' y además no tengan puntos colineales intermedios
    do
    {
        rnd_index_to =
        random_number_generator.Next(0, vertices.Count);
    } while ((rnd_index_from ==
            rnd_index_to) ||
            with_colinear_in_between(vertices, vertices[rnd_index_from], vertices[rnd_index_to]));

    // Tomamos los vértices ubicados en los índices calculados
    from = vertices[rnd_index_from];
    to = vertices[rnd_index_to];

    // Creamos la arista entre dichos vértices
    candidate_edge = new Edge(from, to, Config.color_Edge_default);

    // Se inserta arista si no hay intersección con otra arista de la triangulación
Listing B.3: Source code for the Random triangulation algorithm.
Appendix C

Dominating set Source Code

C.1 Greedy algorithm

```csharp
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using TFG.Configuration;
using TFG.Graph;

namespace TFG.Dominations
{
    class Minimal_Dom : A_Dom
    {
        public override void Algorithm(List<Node> vertices, ref List<Node> dom_nodes)
        {
            List<Node> vertices_c = new List<Node>();

            foreach (Node node in vertices)
            {
                vertices_c.Add(node);
            }

            List<Node> vertices_c = new List<Node>();

            foreach (Node node in vertices)
            {
                vertices_c.Add(node);
            }

            // Mientras que la dominación no sea completa
            do
            {
                // Toma el vértice de mayor grado
                // Lo marca como dominante a él y a
dominados él y sus vecinos
                // Los demás vértices lo borran como vecino
                // Se añade el vértice al conjunto dominante
            }
```
C.2 Connected algorithm

```csharp
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using TFG.Configuration;
using TFG.Graph;

namespace TFG.Dominations
{
    class Conected_Dom : A_Dom
    {
        public override void Algorithm(List<Node> vertices, ref List<Node> dom_nodes)
        {
            // Lista de vértices candidatos a vértices dominantes
            List<Node> candidates = new List<Node>();

            // Vértice dominante en cada iteración
            Node dominating;

            // Primera iteración
            // Toma el vértice de mayor grado
            dominating = greatest_degree(vertices);

            // Lo marca como dominante a él y a dominados él y sus vecinos
            dominating.dominates();

            // Se añaden los vecinos del vértice como candidatos a ser dominantes
            foreach (Node node in dominating.neigh)
            {
                candidates.Add(node);
            }
        }
    }
}
```

Listing C.1: Source code for Greedy dominating set algorithm.
APPENDIX C. DOMINATING SET SOURCE CODE

```csharp
// Los demás vértices lo borran como vecino suyo
dominating.remove_this_from_neighs();

// Se añade el vértice al conjunto dominante
dom_nodes.Add(dominating);

// Mientras que la dominación no sea completa
while (!domination_complete(vertices))
{
    // Toma el vértice de mayor grado
dominating = greatest_degree(candidates);

    // Lo marca como dominante a él y a
dominados él y sus vecinos
dominating.dominates();

    // Se añaden los vecinos del vértice como
candidatos a ser dominantes
foreach (Node node in dominating.neigh)
{
    candidates.Add(node);
}

    // Se elimina el vértice de la lista de
posibles vértices dominantes
candidates.Remove(dominating);

    // Los demás vértices lo borran como vecino suyo
    dominating.remove_this_from_neighs();

    // Se añade el vértice al conjunto dominante
    dom_nodes.Add(dominating);
}
```

Listing C.2: Source code for Connected dominating set algorithm.

C.3 Independant algorithm

```csharp
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
```
C.3. Independant algorithm

```csharp
using TFG.Configuration;
using TFG.Graph;

namespace TFG.Dominations
{
    class Indep_Dom : A_Dom
    {
        public override void Algorithm(List<Node> vertices, ref List<Node> dom_nodes)
        {
            // Lista de vértices candidatos a vértices dominantes
            List<Node> candidates = new List<Node>();
            foreach (Node node in vertices)
            {
                candidates.Add(node);
            }

            // Vértice dominante en cada iteración
            Node dominating;

            // Lista auxiliar con los vecinos del vértice dominante en cada iteración
            List<Node> dom_neighs = new List<Node>();

            // Mientras que la dominación no sea completa
            do
            {
                // Toma el vértice de mayor grado
                dominating = greatest_degree(candidates);

                // Lo marca como dominante a él y a dominados él y sus vecinos
                dominating.dominates();

                // Guardamos los vecinos del vértice dominante en esta iteración
                dom_neighs = dominating.neigh;

                // Se elimina el vértice dominante de la lista de posibles vértices dominantes
                candidates.Remove(dominating);

                // Los demás vértices lo borran como vecino suyo
                dominating.remove_this_from_neighs();

                // Se eliminan los vecinos del vértice dominante como candidatos
            }
        }
    }
}
```
```csharp
// Tambien los demas vertices los eliminan como vecinos suyos
foreach (Node node in dom_neighs)
{
    candidates.Remove(node);
    node.remove_this_from_neighs();
}

// Se añade el vertice al conjunto dominante
dom_nodes.Add(dominating);

} while (!domination_complete(vertices));
```
Appendix D

Experimental Results

D.1 Excel programming

```csharp
// File > automatic execution
private void automaticExecutionToolStripMenuItem_Click(object sender, EventArgs e)
{
    (...)

    // Si se ha aceptado una ejecución automática
    if (dialogresult == DialogResult.OK)
    {
        // Save file dialog
        SaveFileDialog savefile = new SaveFileDialog();

        // set a default file name
        savefile.FileName = "unknown.xlsx";

        // set filters - this can be done in properties as well
        savefile.Filter = "Spreadsheet files (*.xlsx)|*.xlsx";
        savefile.Title = "Export Excel File To";
        savefile.RestoreDirectory = true;

        // Si se ha especificado un archivo
        if (savefile.ShowDialog() == DialogResult.OK)
        {
            string path = savefile.FileName;

            (...)

            // Ventana de loading
            Loading loaderPopup = new Loading();
```
loaderPopup.Show();

// Para las dos triangulaciones
for (int z = 0; z < 2; z++)
{
   (...)

// Para las tres dominaciones
for (int j = 0; j < 3; j++)
{
   // Aplicación excel
   excelApp.DisplayAlerts = false;

   // Se asegura que excel está instalado en el sistema
   if (excelApp == null) {
      MessageBox.Show("Microsoft Excel is not properly installed"); return; }
   (...)

   // Si se va a sobreescribir un archivo
   if (File.Exists(path))
   {
      workbook = excelApp.Workbooks.Open(path);
   }
   else
   {
      workbook = excelApp.Workbooks.Add();
   }
   worksheet = workbook.Worksheets[1];

   worksheet.Cells[3 + (4 * j) + (12 * z), 6] = "Vértices CIERRE CONVEXO";
   worksheet.Cells[5 + (4 * j) + (12 * z), 6] = "Vértices CONJUNTO DOMINANTE";
for (int i = 0; i < automPopup.num_iterations; i++)
{
    this.lienzo.automaticExecution(this, automPopup.num_nodes, 2 + z, j);
    worksheet.Cells[3 + (4 * j) + (12 * z), i + 12] =
        this.lienzo.convexHull_nodes.Count();
    worksheet.Cells[4 + (4 * j) + (12 * z), i + 12] =
        this.lienzo.triang_edges.Count();
    worksheet.Cells[5 + (4 * j) + (12 * z), i + 12] =
        this.lienzo.dominatingSet_nodes.Count();
    if (i % 5 == 0)
    {
        // Dejamos este print para ver si la ejecución se ha congelado al debuggear
        Console.WriteLine("5 iteraciones de "+ dominacion);
    }
}

// Evita fallo cuando no se quiere sobreescribir el archivo
try
{
    workbook.SaveAs(path);
}
Listing D.1: (mainForm.cs) Source code for saving data into an excel file.
D.2 Linear Regression

D.2.1 \( D \) over \( V \)

Figure D.1: Relation between \( D \) and \( V \) for greedy dominating sets under layered triangulations.

Figure D.2: Relation between \( D \) and \( V \) for connected dominating sets under layered triangulations.
Figure D.3: Relation between D and V for independant dominating sets under layered triangulations.

Figure D.4: Relation between D and V for greedy dominating sets under random triangulations.
Figure D.5: Relation between D and V for connected dominating sets under random triangulations.

Figure D.6: Relation between D and V for independant dominating sets under random triangulations.
D.2.2 \( D \) over \( K \)

Figure D.7: Relation between \( D \) and \( V \) for greedy dominating sets under layered triangulations.

Figure D.8: Relation between \( D \) and \( V \) for connected dominating sets under layered triangulations.
D.2. Linear Regression

Figure D.9: Relation between $D$ and $V$ for independant dominating sets under layered triangulations.

Figure D.10: Relation between $D$ and $V$ for greedy dominating sets under random triangulations.
Figure D.11: Relation between D and V for connected dominating sets under random triangulations.

Figure D.12: Relation between D and V for independent dominating sets under random triangulations.
<table>
<thead>
<tr>
<th>Firmante</th>
<th>CN=tfgm.fi.upm.es, OU=CCFI, O=Facultad de Informatica - UPM, C=ES</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fecha/Hora</td>
<td>Sat Jun 11 17:24:57 CEST 2016</td>
</tr>
<tr>
<td>Emisor del Certificado</td>
<td>EMAILADDRESS=<a href="mailto:camanager@fi.upm.es">camanager@fi.upm.es</a>, CN=CA Facultad de Informatica, O=Facultad de Informatica - UPM, C=ES</td>
</tr>
<tr>
<td>Numero de Serie</td>
<td>630</td>
</tr>
<tr>
<td>Metodo</td>
<td>urn:adobe.com:Adobe.PPKLite:adbe.pkcs7.sha1 (Adobe Signature)</td>
</tr>
</tbody>
</table>