Alma Mater Studiorum · Università di Bologna

**SCUOLA DI SCIENZE**

**Corso di Laurea Magistrale in Matematica**

# The isomorphism problem for directed acyclic graphs: an application to multivector fields

**Tesi in topologia computazionale**

**Relatore:**
**Chiar.mo Prof.**
**Massimo Ferri**

**Correlatore:**
**Dott.**
**Mateusz Juda**

**Presentata da:**
**Caterina Tamburini**

**III Sessione**
**Anno Accademico 2016/2017**

Faith for absolute victory.

*Fifth eternal guideline of the Soka Gakkai*

# Introduction

This thesis is based on a project developed by a group of researchers at the Faculty of Mathematics and Computer Science at the Jagiellonian University of Krakow. They have recently introduced a new tool in computational topology: the multivector field. One of the reasons why they created such a tool is to study, from a combinatorial point of view, sampled dynamics. Thus their work starts with a could of vectors which is transformed into a combinatorial multivector field on a simplicial complex. From a combinatorial multivector field it is then possible to construct a directed graph. Now, applying to this directed graph a decomposition into strongly connected components, we obtain a directed acyclic graph, called Morse graph, that describes the global dynamics of the initial cloud of vectors.
So an important question might be: how much a perturbation in the cloud of vectors can affect the global dynamics described by the multivector field? To give an answer to this question we may start comparing Morse graphs. The first step might be to study the isormophism problem, that is we try to understand if two Morse graphs, or more in general two directed acyclic graphs, are equal up to a permutation of their vertices' labels. A second step could be the analysis of how much two not isomorphic graphs are actually different.
This dissertation is focused on the first step: the isomorphism problem. However, at the end we propose a possible way to proceed with the second step.
The main issue regarding the isomorphism of graphs is the fact that it is still an open problem. In particular it is known to be NP, but we still don't know if it is also NP-complete. Moreover, there are no general characterization of isomorphic graphs. We have many invariants which enable us to say if two graphs are definitely not isomorphic,

but no one of these invariants is complete, while for instance a complete invariant exists for trees isomorphism (see [16]).
Therefore to state that two graphs are for sure isomorphic we have to express an isomorphism between them. Fortunately there are many algorithms developed to find an isomorphism between two graphs if it exists.

The work is organized as follows.
In the first chapter we present the general problem. In particular the first section is dedicated to the theory of multivector fields and to how we create a Morse graph. In the second section we define the isomorphism of graphs and we describe one of the algorithms cited above.
In the second chapter we describe four different ways to distinguish not isomorphic directed graphs. The first section aims to prove four simple features, that isomorphic directed graphs share, and each one is translated into a Python script described in the second section. Our propositions are based on a partition of the set of vertices induced by their degree. So in the third section we describe an alternative partition, finer than the one we used.
In the third chapter are several examples. In particular with the first section of this chapter we give examples created specifically for our tests. In the second section we analyze the class of regular acyclic graphs, which are the most difficult to distinguish one another.
The fourth chapter is about a possible way to develop the future work, that is the study of multivector fields as finite topological spaces. Therefore in the first section we give some notions about the algebraic topology of finite topological spaces. The second section describes one possible application of this theory to the comparison of multivector fields.

# Introduzione

Questa tesi si basa su un progetto sviluppato da un gruppo di ricercatori della Facoltà di Matematica ed Informatica dell'Università Jagiellonica a Cracovia. Recentemente loro hanno introdotto un nuovo strumento nell'ambito della topologia computazionale: i campi multivettoriali. Uno dei principali motivi per cui hanno creato un tale strumento è lo studio, da un punto di vista combinatorio, di sistemi dinamici noti solo mediante un campione di dati. Il loro lavoro inizia dunque con una nuvola di vettori che viene trasformata in un campo multivettoriale su un complesso simpliciale. Dal campo multivettoriale si passa poi a definire un grafo diretto. Ora, applicando a questo grafo diretto una decomposizione in componenti fortemente connesse, si ottiene un grafo diretto aciclico, detto grafo di Morse, che descrive la dinamica globale della nuvola di vettori.
Una domanda fondamentale potrebbe quindi essere: quanto una perturbazione nella nuvola di vettori può influire sulla dinamica globale descritta dal campo multivettoriale? Per dare una risposta a questa domanda si può iniziare confrontando i grafi di Morse. Il primo step potrebbe essere studiarne l'isomorfismo, cioè cerchiamo di capire se due grafi di Morse, o più in generale due grafi diretti aciclici, sono uguali a meno di una permutazione delle etichette dei loro vertici. Un secondo step potrebbe essere l'analisi di quanto due grafi non isomorfi, sono effettivamente diversi.
In questa tesi ci si concentra principalmente sul primo step: il problema dell'isomorfismo. Ad ogni modo nell'ultimo capitolo proponiamo una possibile strada da seguire per il secondo.
La principale questione riguardante l'isomorfismo di grafi è il fatto che questo problema è ancora aperto in matematica. In particolare si sa che è un problema della classe NP ma non è ancora noto se sia NP-completo o meno. Inoltre non esistono caratterizzazioni

per i grafi isomorfi. Ci sono molti invarianti che ci permettono di stabilire se due grafi sono definitivamente non isomorfi, ma nessuno di essi è un invariante completo, mentre ad esempio esiste un invariante completo per l'isomorfismo di alberi (si veda [16]).
Di conseguenza, per stabilire con certezza che due grafi sono isomorfi bisogna esplicitarne un isomorfismo. Fortunatamente ci sono molti algoritmi sviluppati per trovare un isomorfismo tra due grafi, se esso esiste.

Il lavoro è organizzato come segue.
Nel primo capitolo presentiamo il problema generale. Il primo paragrafo è dedicato alla teoria dei campi multivettoriali e a come viene creato un grafo di Morse. Nel secondo paragrafo definiamo l'isomorfismo tra grafi e descriviamo uno degli algoritmi citati sopra.
Nel secondo capitolo sono descritti quattro diversi modi per distinguere grafi diretti non isomorfi. Il primo paragrafo mira a dimostrare quattro semplici proprietà che grafi diretti isomorfi condividono ed ognuna è tradotta in uno script Python descritto nel secondo paragrafo. Le nostre proposizioni si basano su una partizione dell'insieme dei vertici indotta dal loro grado. Nel terzo paragrafo descriviamo quindi una partizione alternativa, più fine di quella usata in precedenza.
Nel terzo capitolo vi sono svariati esempi. In particolare con il primo paragrafo di questo capitolo presentiamo degli esempi creati appositamente per i nostri test, mentre nel secondo analizziamo la classe dei grafi aciclici regolari, che sono i più difficili da distinguere l'un l'altro.
Il quarto capitolo riguarda infine una possibilità per sviluppare il lavoro futuro, che è lo studio dei campi multivettoriali come spazi topologici finiti. Quindi nel primo paragrafo presentiamo alcune nozioni di topologia algebrica nell'ambito degli spazi topologici finiti.
Il secondo paragrafo descrive una possibile applicazione di questa teoria al confronto dei campi multivettoriali.

# Contents

# Chapter 1

# Isomorphism of Morse graphs

In this first chapter we present the problem analyzed throughout the whole work and a tool to solve it that already exists.
Main references for the first section are [4] and [6]. Our definitions are taken from [4], but in [6] are given more general ones. Moreover in this section we cite cubical and cell complexes; for a deepened treatise of the former we refer to [5] and for a definition of the latter to [13].
The second section is dedicated to *Networkx*, a Python library created to work with graphs and networks, whose documentation [9] is available on internet.

## 1.1 The problem

The purpose of this section is to describe how the object of our study, i.e. Morse graphs, is constructed in the specific context of multivector fields.
Given a cloud of vectors we create on it a simplicial mesh and then a multivector field, which itself describes a dynamics for the cloud of vectors. Therefore we have a combinatorial dynamical system, which may be visualized as a directed graph. As a consequence, the term dynamics may be translated in the concept of all paths of a directed graph, both finite and infinite.
At this point we extract from the directed graph the recurrent dynamics studying how

its vertices tend to group together into clusters. These clusters are called strongly connected components. There are three main reasons why the study of such components is useful. First of all, the set of strongly connected components is efficiently computable. Moreover examining strongly connected components lends itself to extracting rigorous results when considering directed graphs obtained from continuous processes. And as last but central feature, we have that strongly connected components describe the global dynamics of the directed graph. In fact, collapsing each strongly connected component to a single vertex and using a partial order upon them, it is possible to create a directed acyclic graph, namely the Morse graph, which is a description of the global dynamics of the system.
Our task is to study the isomorphism of directed acyclic graphs because we want to compare Morse graphs. Since they describe global dynamics, to have similar Morse graphs means to have similar global dynamics.

### 1.1.1 Lefschetz complex

We start our description with a cloud of vectors because behind the mathematical model presented in [4] there is the analysis of sampled dynamics, that is dynamics known only from a sample.
The sampled data often consists of a **cloud of vectors**, which is

$$V := \left\{\vec{v} = (s_v, t_v) | s_v, t_v \in \mathbb{R}^d\right\}.$$

The first step we do in order to analyze the cloud of vectors, is the creation of a simplicial mesh $X$ such that $s_v$ is a vertex of $X$.
We remind that a **simplicial complex** is a family $X$ of **simplices**, that are non-empty subsets of a finite set of vertices, such that any non-empty subset $\sigma$ of a simplex $\tau \in X$, called a **face** of $\tau$, is in $X$.
In our examples we use simplicial complexes but actually one might use cubical or cell complexes. More in general, the mesh can be created using a Lefschetz complex, which is a notion that generalizes all the previous.

**Definition 1.1.** Let $\mathcal{R}$ be a ring with unity. $(X, k)$ is a **Lefschetz complex** if $X = (X_q)_{q\in\mathbb{Z}^+}$ is a finite set with gradation, $k : X \times X \to \mathcal{R}$ is a map such that $k(x, y) \neq 0$

implies $x \in X_q$, $y \in X_{q-1}$ and for any $x, z \in X$ we have:

$$\sum_{y \in X} k(x,y)k(y,z) = 0 \tag{1.1}$$

The elements of $X$ are called **cells** and $k(x,y)$ is the **incidence coefficient** of $x$ and $y$.

Some simple examples of Lefschetz complexes are provided not only by the family of all simplexes of a simplicial complex, as we said above, but also by the family of all cubes in a cubical complex or a regular finite CW-complex. In these cases the incidence coefficient is obtained from the boundary homomorphism of the associated simplicial, cubical or cellular chain complex.

Since we give examples using simplicial complexes we write the expression of the map $k$ in this particular case, as it is defined in [15].

A simplex $\sigma$ of a simplicial complex $X$ is coded as $\sigma =< v_0, \dots, v_q >$ where $v_i$ are 0-simplexes of $\sigma$ for $i = 1, \dots, n$. The map $k$ is

$$k(\sigma, \tau) := \begin{cases} (-1)^i & \text{for } \sigma =< v_0, \dots, v_q > \text{ and } \tau =< v_0, \dots, v_{i-1}, v_{i+1}, \dots, v_q > \\ 0 & \text{otherwise} \end{cases}$$

Thanks to 1.1 we have a free chain complex $(\mathcal{R}(X), \partial^k)$ where $\mathcal{R}(X)$ is the free group spanned by $X$ and $\partial^k : \mathcal{R}(X) \to \mathcal{R}(X)$ is defined on generators by

$$\partial^k(x) := \sum_{y \in X} k(x,y)y.$$

The **Lefschetz homology** of $(X, k)$, denoted by $H^k(X)$, is the homology of this chain complex.

It is possible to define a relationship among the cells of a Lefschetz complex.

Given two cells $x, y \in X$ we say that $y$ is a **facet** of $x$ and write $y \prec_k x$ if $k(x,y) \neq 0$. The relation $\prec_k$ extends uniquely to a minimal partial order, which is denoted by $\leq_k$. We say that $y$ is a **face** of $x$ if $y \leq_k x$.

**Definition 1.2.** The **closure** of $A \subseteq X$, denoted $clA$, is obtained by recursively adding to $A$ the facets of cells in $A$, the facets of the facets of cells in $A$ and so on.

The set $A$ is **closed** if $clA = A$.

The **mouth** of $A$ is $moA := clA \setminus A$.
The set $A$ is said to be **proper** if $moA$ is closed.

*Remark* 1. Every proper subset of a Lefschetz complex with incidence coefficient restricted to this subset is a Lefschetz complex too. So all the previous definitions and constructions are suitable for proper subsets of a Lefschetz complex.

We now present an example to sum up all the definitions given so far.

**Example 1.1.** In the simplicial complex $K$ in fig 1.1 we can consider the subset $A$ composed by the cells $\{2, 3, 4\}$. The closure of $A$ is $cl(A) = \{0, 1, 2, 3, 4, 5, 6\}$ and its mouth is $mo(A) = \{0, 1, 5, 6\}$.
An example of not closed subset of $K$ is $mo(A)$, while $B = \{7\}$ is closed.
The last subset marked in figure 1.1, that is $C = \{11, 12, 13, 14\}$, is an example of proper subset. Indeed for its closure we have

$$cl(C) = \{8, 9, 10, 11, 12, 13, 14\}$$

so $mo(C) = \{8, 9, 10\}$ and it is closed.

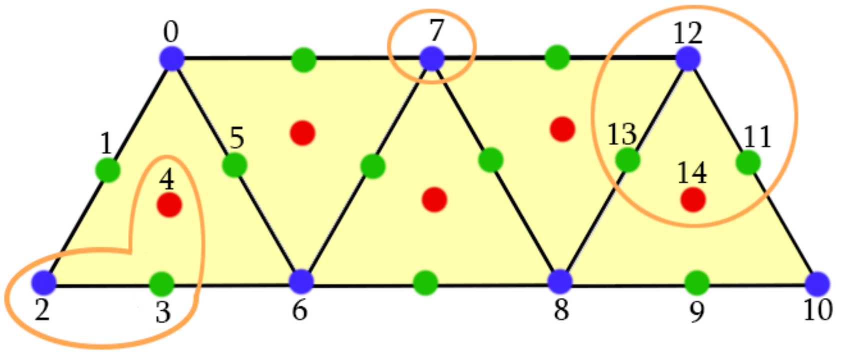

Figure 1.1: A simplicial complex consisting of 7 0-simplexes, indicated with blue dots, 11 1-simplexes or edges, each one with a green dot, and 5 2-simplexes, indicated with red dots

### 1.1.2 Combinatorial multivector fields

Given the simplicial complex with the cloud's vectors attached to all its vertices, the second step we do in our analysis, is the creation of a combinatorial multivector field. For a deepened description of this delicate step we refer to [6]. Here we simply want to give the definitions that are necessary to create the Morse graph.
To keep the treatise more general, in the following, $(X, k)$ is a fixed Lefschetz complex. Moreover we need another fundamental notion before defining the combinatorial multivector field.

**Definition 1.3.** A **combinatorial multivector** or briefly a multivector, is a proper subset $V \subseteq X$ admitting a unique maximal element with respect to the partial order $\leq_k$. This element is called **dominant** and, from now on, it is denoted $V^*$.

As a consequence of remark 1, we can calculate the Lefschetz homology of a multivector.
A multivector $V$ is said to be **regular** if its Lefschetz homology is zero; otherwise it is called **critical**. Trying to give an intuitive vision of a regular multivector $V$, we can say it is such that $clV$ may be collapsed to $moV$. On the contrary a critical multivector indicates that $clV$ may not be collapsed to $moV$.
Now we can introduce the main concept described in [4].

**Definition 1.4.** A **combinatorial multivector field**, or briefly a multivector field, on $X$ is a partition $\mathcal{V}$ of $X$ into multivectors.

For each cell $x \in X$ we denote by $[x]_{\mathcal{V}}$ the unique multivector in $\mathcal{V}$ to which $x$ belongs and $x^*$ the dominant cell of $[x]_{\mathcal{V}}$.

**Example 1.2.** In figure 1.2 we have a partition of the simplexes of a simplicial complex $K$ in which every simplex is indicated with a dot in its center of mass. Similarly to example 1.1, 0-simplexes are marked with blue dots, 1-simplexes are marked with green dots and 2-simplexes with orange dots. The partition's blocks are indicated with red ovals both solid and dashed. This partition is not a multivector field because the block $\{0, 1, 2\}$ is not a proper subset of $K$, so it is not a multivector. The other block marked with a dashed oval, namely $\{3, 4, 5\}$, is a proper subset but it has two maximal elements, so again it is not a multivector.

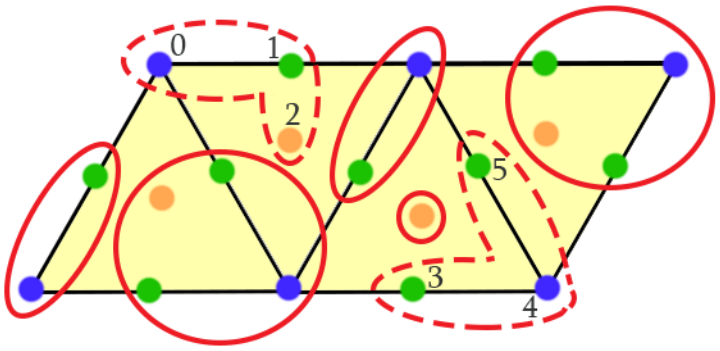

Figure 1.2: A partition of a Lefschetz complex that is not a multivector field

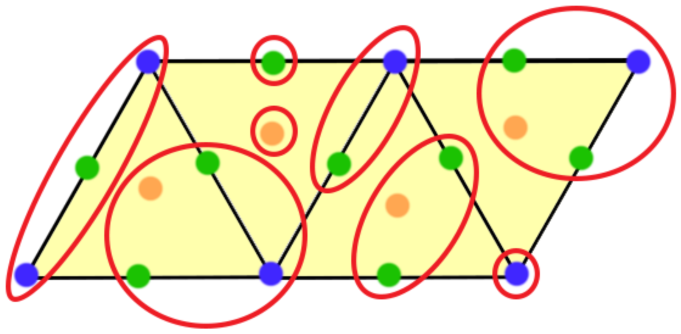

Figure 1.3: A multivector field over the same complex given in 1.2

In figure 1.3 we have an example of combinatorial multivector field.
This multivector field presents many examples of critical multivectors, for instance those with only one element are all critical.

*Remark* 2. We may notice that it is always possible to define a multivector field on a Lefschetz complex. For instance the multivector field in which every element of the partition is a singleton always exists.

### 1.1.3 Dynamics on multivector fields

A **combinatorial dynamical system** on a simplicial complex $X$ is a multivalued map $F : X \rightrightarrows X$, i.e. a map $F : X \to \mathcal{P}(X)$, that sends every simplex of $X$ into a family of simplexes in $X$. Since the concept of multivalued map is equivalent, on the formal level, to the concept of directed graph, we can see a combinatorial dynamical system $F$ as a directed graph $G_F$ whose vertices are simplexes of $X$ and with a directed edge from $x$ to $y$ if and only if $y \in F(x)$. A direct consequence of this interpretation of a combinatorial dynamical system as a directed graph, is that some concepts in dynamics may be translated into concepts in directed graphs and vice versa.
Now we can go back to our multivector field to show how we can create from it a directed graph, and so a multivalued map. Again we use the more general notion of Lefschetz complex instead of simplicial complex. Moreover we remind that the dominant cell of a multivector $V$ is denoted $V^*$.
Given a multivector field $\mathcal{V}$ on a Lefschetz complex $X$, we associate with it a dynamics using a directed graph $G_{\mathcal{V}}$. This graph has vertices in $X$ and three types of arrows: **up-arrows**, which have heads in $V^*$ and tails in all the other cells of $V$, **down-arrows**, with tails in $V^*$ and heads in $moV$, and **loops**, that join $V^*$ with itself for all critical multivectors $V$. If we want to give a more formal definition of the arcs of $G_{\mathcal{V}}$, we can say that there is an arrow from a cell $x$ to a cell $y$ if one of the following conditions is satisfied:

1. $x \neq y = x^*$ (up-arrow)
2. $x = x^*$and $y \in clx \setminus [x]_{\mathcal{V}}$ (down-arrow)
3. $x = x^* = y$ and $[y]$ is critical (loop)

A useful observation we could do is that the up-arrows sharing the same head uniquely determine a multivector.
An example of directed graph associated with a multivector field is in figure 1.4. Here up-arrows are the solid ones while down-arrows are dashed. The loops are indicated with a red circle around the cell itself.

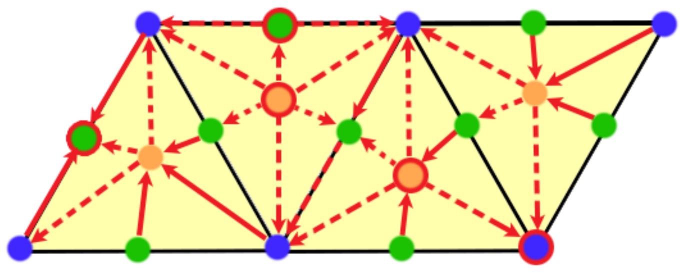

Figure 1.4: The graph $G_{\mathcal{V}}$ associated with the multivector field in 1.3

It is possible to define a relation on $X$ given by the arrows of $G_{\mathcal{V}}$. We can write $y \prec_{\mathcal{V}} x$ if there is an arrow from $x$ to $y$ in $G_{\mathcal{V}}$.
So in our case we can define the multivalued map $\Pi_{\mathcal{V}} : X \rightrightarrows X$ which sends a cell $x$ to the set

$$\Pi_{\mathcal{V}}(x) := \{y \in X | y \prec_{\mathcal{V}} x\}.$$

### 1.1.4 Strongly connected components

Now we want to extract from this directed graph the family of its strongly connected components. For the sake of clarity we give some definitions and an example, taken from [14], regarding only directed graphs, before describing the decomposition in the context of multivector fields.
A **strongly connected component** of a directed graph $G$ is a maximal subset $C$ of the set of vertices such that for every pair of vertices $u, v \in C$ there is a directed path from $v$ to $u$ and a directed path from $u$ to $v$. The family of the strongly connected components of a directed graph is naturally ordered. In fact, given an index set $P$ for the collection of strongly connected components, we can define a partial order saying that, for $p, q \in P$, $q \leq p$ if there is a path in $G$ starting from a vertex in $q$ and ending at a vertex in $p$.
From this partially ordered family, one can construct a new graph, which results directed acyclic, by collapsing each strongly connected component to a single vertex and forming an edge $q \to p$ if $q \leq p$.

**Example 1.3.** Let the one in figure 1.5 be our directed graph. We can think at this graph as a model for the propagation of information on a social network. From this point of view each state correspond to the users who are in possession of the information. In a basic setting we may consider $n$ users, which correspond to $n$ vertices, and if user $i$ has access to the information of user $j$, then there is a directed edge $j \to i$.

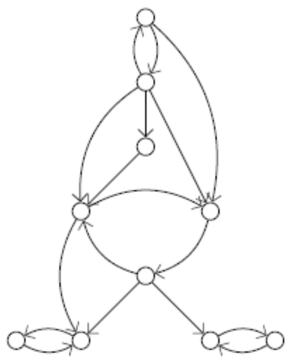

Figure 1.5: Directed graph

This graphs contains four strongly connected components and only one vertex does not lie in any component. Indeed the information can be shared among users in the same component but it can only be propagated further to other users according to the partial order on the strongly connected components. The Morse graph reflects this idea.

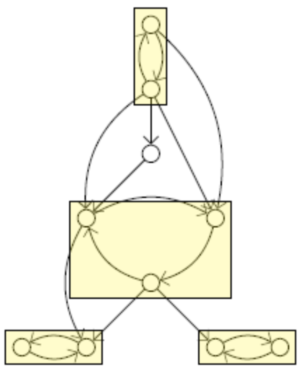

Figure 1.6: Strongly connected path components in 1.5

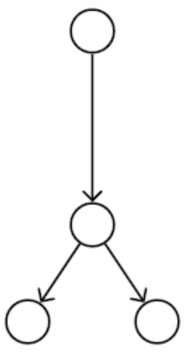

Figure 1.7: Morse graph of 1.5

### 1.1.5 Morse graph

With this paragraph we translate the idea of strongly connected component in a similar concept for the multivector fields. Moreover we describe a more general decomposition adapting some notions of dynamical systems to the context of multivector fields. Again we start with a directed graph, but here the decomposition of $G_{\mathcal{V}}$ in strongly connected components is made by means of basic sets and developed as follows.
Let $A \subseteq X$ be $\mathcal{V}$-compatible, i.e $A$ equals the union of multivectors contained in it. We write $x \overset{A}{\to}_{\mathcal{V}} y$ if there exists a path of $G_{\mathcal{V}}$ in $A$ from $x^*$ to $y^*$ of length at least one. We also write $x \overset{A}{\leftrightarrow}_{\mathcal{V}} y$ if $x \overset{A}{\to}_{\mathcal{V}} y$ and $y \overset{A}{\to}_{\mathcal{V}} x$.
The subset $A$ is said to be **weakly recurrent** if for every $x \in A$ we have $x \overset{A}{\leftrightarrow}_{\mathcal{V}} x$. And it is **strongly recurrent** if for any $x, y \in A$ we have $x \overset{A}{\leftrightarrow}_{\mathcal{V}} y$. The **chain recurrent set** of $X$ is the maximal weakly recurrent subset of X and it is denoted $CR(X)$. Therefore

$$CR(X) := \left\{ x \in X | x \overset{X}{\leftrightarrow}_{\mathcal{V}} x \right\}$$

The relation $\leftrightarrow_{\mathcal{V}}$ is an equivalence relation when restricted to $CR(X)$ so it is possible to consider its equivalence classes, we call them **basic sets** of $\mathcal{V}$.
Given two basic sets $B_1, B_2$ we write $B_1 \leq_{\mathcal{V}} B_2$ if there exists a path in $G_{\mathcal{V}}$ such that all its sufficiently early elements belong to $B_2$ and all its sufficiently far elements belong to $B_1$. The relation $\leq_{\mathcal{V}}$ is a partial order on the family of basic sets. The Hasse diagram of this partial order is our **Morse graph**.

*Remark* 3. The family of basic sets with the partial order just defined is an example of Morse decomposition. Even though we would like to give only an intuitive definition of it, we need to describe some more notions.
A **solution** of a multivector field $\mathcal{V}$ is a sequence of cells such that any two consecutive cells in it form an arrow in the graph $G_{\mathcal{V}}$. We may have different types of solutions. If the sequence is bi-infinite the solution is said to be **full**. But the sequence could be also backward infinite, forward infinite and finite.
A subset $A \subseteq X$ is said to be **invariant** if for every multivector $V \subseteq A$ there is a full solution through $V^*$ in $A$.
A finite solution in $clA$ is an **internal tangency** in $A$ if the values at its endpoints are in $A$ but one of the other values is not in $A$.

A subset $A \subseteq X$ is an **isolated invariant set** if it is invariant and admits no internal tangencies.
Finally we can give an informal definition of Morse decomposition.
Given a family $\mathcal{M} = \{M_r\}_{r \in \mathbb{P}}$ of mutually disjoint, non-empty, isolated invariant sets, we write $r \leq r'$ for $r, r' \in \mathbb{P}$ if there exists a full solution such that all its sufficiently far terms belong to $M_r$ and all sufficiently early terms belong to $M_{r'}$. We say that the family $\mathcal{M}$ is a **Morse decomposition** of $X$ if the relation $\leq$ on $\mathbb{P}$ is a partial order.
Again the Morse graph is the Hasse diagram of the partial order $\leq$.
As it is proved in [4, Theorem 8.4], every basic set is a strongly recurrent isolated invariant set.

We conclude this section with an example regarding a multivector field.

**Example 1.4.** In figure 1.8 we have a simplicial complex $K$ with four 0-simplexes, six 1-simplexes and three 2-simplexes. The multivector field $\mathcal{V}$ is defined by the associated graph $G_{\mathcal{V}}$; only up-arrows and loop are drawn in figure 1.8. Cells sharing the same number are contained in the same basic set because they are all equivalent by means of $\leftrightarrow_{\mathcal{V}}$. So they provide three different blocks in the Morse decomposition of $K$.

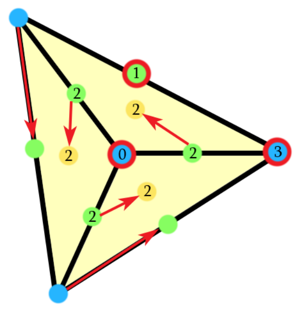

Figure 1.8: Morse decomposition of a multivector field. Cells with same number are in the same element of the decomposition.

In figure 1.9 we have the Morse graph of 1.8. If we want to say something about the global dynamics of $\mathcal{V}$ we have to remind two notions. An isolated invariant set is an **attractor**, respectively a **repeller**, if there aren't full solutions crossing it which go

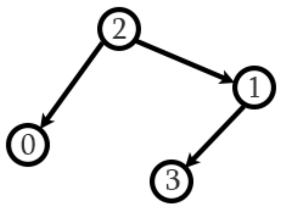

Figure 1.9: The Morse graph of the Morse decomposition in 1.8.

away from it forward in time, respectively backward in time.
From the Morse graph in figure 1.9 we can understand that the dynamics created by $\mathcal{V}$ has a repeller, that is the vertex with label 2, two attractors that are 0 and 3, and an isolated invariant set which is neither an attractor nor a repeller.

## 1.2 An algorithm for graphs isomorphism

As we said in the previous section, our target is to compare Morse graphs. To keep the topic more general, since Morse graphs are directed acyclic, we want to understand whether two directed acyclic graphs are isomorphic or not.
We recall the definition of graph isomorphism, as it is presented in [1] and [2].

**Definition 1.5.** Two directed graphs $G = (V, A)$ and $G' = (V', A')$ are **isomorphic** if there exist two bijections $\phi : V \to V'$ and $\psi : A \to A'$ such that $x$ is an arc from $u$ to $v$ in $G$ if and only if $\psi(x)$ is an arc from $\phi(u)$ to $\phi(v)$ in $G'$.
The pair of functions $(\phi; \psi)$ is called an **isomorphism**.

Unfortunately the isomorphism problem is still an open problem in mathematics. It belongs to the class NP but we still don't know if it is NP-complete. Nevertheless there are many algorithms to find an isomorphism between two graphs, if it exists.
An example of two isomorphic graphs is given by $G_1$ in figure 1.10 and $G_2$ in 1.11.

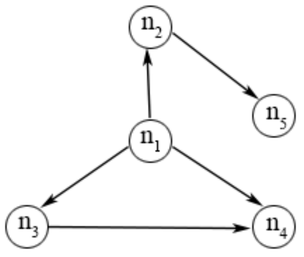

Figure 1.10: Graph $G_1$

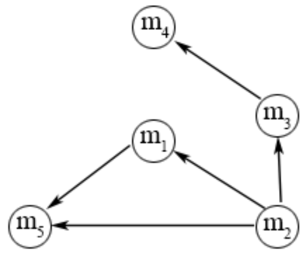

Figure 1.11: Graph $G_2$

### 1.2.1 Networkx

In this section we are going to describe one of the algorithms for graph isomorphism. It is implemented in a library called *Networkx*.
Networkx is a Python package for the creation, study and manipulation of the structure, dynamics and functions of complex networks. There are four basic Python classes to implement undirected simple graphs, directed graphs, undirected multi-graphs and directed multi-graphs. The graph internal data structures are based on an adjacency list representation and implemented using Python dictionary datastructures. The adjacency structure, implemented as a Python dictionary of dictionaries, allows fast addition, deletion, and lookup of nodes and neighbors in large graphs.
Moreover this library provides us many algorithms to study and compare graphs. Going back to our target, since the object of our study are simply directed acyclic graphs we can convert them into a Networkx's graph-like object and analyze the isomorphism using the function *is-isomorphic*. This function is the only one in the package that can give us a positive answer; it returns *True* if the graphs given as parameters are isomorphic and returns *False* otherwise. There are three more functions to compare graphs but they cannot give a positive answer. They simply return *False* if the two graphs are definitely not isomorphic, that is exactly what our tests, presented in the following chapter, are supposed to do.
The function *is-isomorphic* is based on a matching algorithm presented in [7]. It is called `VF2` and is an improved version of the `VF` algorithm created by the same authors. In our presentation we refer to both [7] and [8]. We suppose that the analyzed graphs are di-

rected and we refer to them as $G_1 = (V_1, A_1)$ and $G_2 = (V_2, A_2)$; anyway the extension of the algorithm to undirected graphs is trivial.
The algorithm tries to create a matching between two graphs, that is equivalent to trying to express an isomorphism.
A matching process between two graphs $G_1$ and $G_2$ consists in the determination of a mapping $M$ which associates nodes of the first graph to nodes of the second one and vice versa. The mapping $M$ is often expressed as the set of ordered pairs $(n, m)$, with $n \in V_1$ and $m \in V_2$, each representing the mapping of the node $n$ into the node $m$. So we simply have

$$M = \{(n, m) \in V_1 \times V_2 | n \text{ is mapped onto } m\}.$$

The matching process can be suitably described by means of a sequence of states. Each state $s$ of the matching process can be associated with a partial matching solution $M(s)$, that is a subset of $M$ so it contains only some components of $M$. A partial mapping solution univocally identifies two subgraphs $G_1(s)$ of $G_1$ and $G_2(s)$ of $G_2$ obtained by selecting from $G_1$ and $G_2$ only the nodes included in the components of $M(s)$ and the edges connecting them.
In 1.5 there is an example of partial matching $M(s)$ with $G_1(s)$ and $G_2(s)$. We write $M_1(s)$ and $M_2(s)$ to indicate the projections of $M(s)$ into $V_1$ and $V_2$ respectively.
Looking at a partial matching as a state of the matching process, we have that the transition between two states corresponds to the addition of a new pair of matched nodes.

**Example 1.5.** Here we present an example of a matching between two graphs $G_1$, in fig 1.10, and $G_2$, in fig 1.11.
The only possible matching between these two graphs is

$$M = \{(n_1, m_2), (n_2, m_3), (n_5, m_4), (n_3, m_1), (n_4, m_5)\}$$

We can consider a state $s$ such that $M(s) = \{(n_1, m_2), (n_4, m_5)\}$. The graphs associated with this state are in figure 1.12.

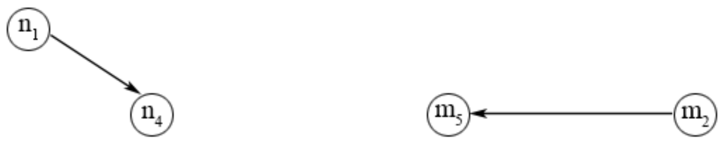

Figure 1.12: On the left the graph $G_1(s)$ and on the right the graph $G_2(s)$

To visualize the transition between two states of the matching process, we now consider a state $s'$ obtained from $s$ adding the pair $(n_3, m_1)$. The figure 1.13 shows the transition. Among all the possibilities from the state $s$, represented as arrows with tail in $s$, we choose the one that brings us to $s'$.

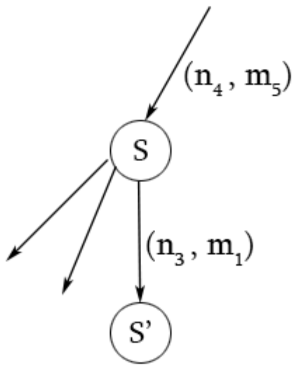

Figure 1.13: A representation of the transition between two states of the matching process.

Since we changed state we had a development in the matching process, so there are two new graphs associated with $s'$. They are shown in figure 1.14

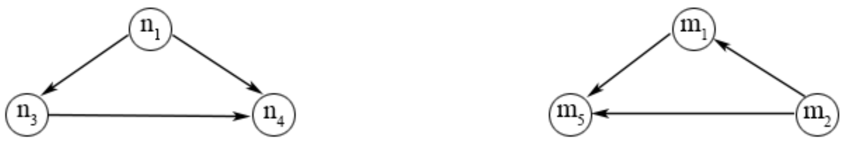

Figure 1.14: On the left the graph $G_1(s')$ and on the right the graph $G_2(s')$

As we saw, the final matching is created adding matched nodes to a partial solution. Obviously there are always many possibility for adding a pair of vertices to a partial matching. In order to reduce the number of these possibilities, we impose that the corresponding partial solution verifies some coherence conditions, depending on the desired mapping type. For instance, to have an isomorphism it is necessary that the partial mappings are isomorphisms between the corresponding subgraphs. If the addition of a pair to the partial mapping produces a state that is in contradiction with the coherence condition, we may ignore it because it certainly cannot lead to the final goal. For a given state $s$, in the algorithm are introduced some criteria for foreseeing if $s$ has no coherent successors after a certain number of steps. These criteria are called **feasibility rules**; in particular a rule implements a **k-look-ahead** if, given a state $s$ and a pair $(n, m)$ to be included in $s$ to create a new state $s'$, it allows us to establish if all the states reachable from $s'$ in $k$ steps are incoherent. The duty to check these rules is given to a **feasibility function** that is a boolean function $F(s, n, m)$. It returns *True* if it is guaranteed that the state $s'$ obtained from $s$ adding $(n, m)$, is a partial isomorphism whenever $s$ is. Therefore, at the end of the process the final state is either an isomorphism between $G_1$ and $G_2$ or a graph-subgraph isomorphism between a subgraph of $G_1$ and $G_2$.
Now we give the outline of the algorithm.

**Algorithm 1** VF2 Algorithm

```
procedure MATCH($s$)
    INPUT: an intermediate state s; the initial state $s_0$ has $M(s_0) = \emptyset$
    OUTPUT: the mapping between the two graphs
    if $M(s)$ covers all the nodes of $G_2$ then OUTPUT $M(s)$
    else
        Compute the set $P(s)$ of the pairs candidate for inclusion in $M(s)$
        for all $(n, m) \in P(s)$ do
            if $F(s, n, m)$ then
                Compute the state $s'$ obtained by adding $(n, m)$ to $M(s)$
                CALL Match($s'$)
            end if
        end for
        Restore data structures
    end if
end procedure
```

The definition of the set $P(s)$ containing the node pairs that can be added to the current state is based on the definition of two other sets. They are the **out-terminal set** $T_i^{out}(s)$ and the **in-terminal set** $T_i^{in}(s)$ for $i = 1, 2$, defined as follows.

$$T_i^{out}(s) = \{v \in V_i | v \notin M_i(s) \text{ but } \exists (n, v) \in A_i \text{ for some } n \in M_i(s)\} \tag{1.2}$$

$$T_i^{in}(s) = \{v \in V_i | v \notin M_i(s) \text{ but } \exists (v, n) \in A_i \text{ for some } n \in M_i(s)\} \tag{1.3}$$

Now we are able to construct $P(s)$; we have three possibilities to do it.
If both $T_1^{out}(s)$ and $T_2^{out}(s)$ are not empty then

$$P(s) = T_1^{out}(s) \times \left\{\min T_2^{out}(s)\right\}$$

where $\min T_2^{out}(s)$ is the vertex in $T_2^{out}(s)$ with smallest label, or if another type of total order is fixed, it is the smallest vertex in that order.
If both $T_1^{out}(s)$ and $T_2^{out}(s)$ are empty, and both $T_1^{in}(s)$ and $T_2^{in}(s)$ are not, then

$$P(s) = T_1^{in}(s) \times \left\{\min T_2^{in}(s)\right\}.$$

If all the four terminal sets are empty, then

$$P(s) = (V_1 \setminus M_1(s)) \times \{\min(V_2 \setminus M_2(s))\} .$$

In the case that only one of the in-terminal sets or only one of the out-terminal sets is empty, it may be proved that the state $s$ cannot be part of a matching, and it is no further explored.

To conclude this section we would like to turn again our attention to the feasibility function. As we said before it implements the feasibility rules. In the specific case of an exact graph isomorphism, the rules are of three types: 0-look-ahead, 1-look-ahead and 2-look-ahead. Therefore to evaluate $F(s, n, m)$ the algorithm 1 examines all the nodes connected to $n$ and $m$; if such nodes are in the current partial mapping the algorithm checks if each branch from or to $n$ has a corresponding branch from or to $m$ and vice versa, that is a 0-look-ahead rule. Otherwise, if these nodes are not in the current partial matching, the algorithm 1 counts how many nodes are in $T_i^{in}(s)$, $T_i^{out}(s)$ and $(V_i \setminus M_i(s) \setminus T_i^{in}(s) \setminus T_i^{out}(s))$. For the isomorphism these counts must be equals for $n$ and $m$, moreover they correspond to a rule of the type 1-look-ahead and 2-look-ahead respectively.

A last observation could be done about feasibility rules.

There are two kinds of feasibility rules, those described above regard the *syntax* of the graphs. The other type concerns the *semantics*.

If the nodes and the branches of the graphs being matched also carry semantic attributes, another condition must hold for $F(s, n, m)$ to be valuated as *True*. Namely the attributes of the nodes and the branches paired must be compatible. The semantic compatibility has to be defined with reference to the specific application domain.

# Chapter 2

# Four negative answers

In this chapter we present our tests to establish whether two directed acyclic graphs are not isomorphic.
Main references for the definitions in the first section are [2] and [1].
In the second section we briefly describe the implementation of our tests. The complete code can be found in the appendix A.
The third section is dedicated to a particular partition of the set of vertices introduced in [3].

## 2.1 Four properties of isomorphic directed graphs

If two graphs are isomorphic, they necessarily share some particular features. Conversely if two graphs don't share one of these particular features, they for sure cannot be isomorphic.
Here we analyze four of these properties, each one is presented by a proposition and leads to a Python test described in the next section.
When two directed graphs, or briefly digraphs, $G$ and $G'$ are isomorphic we write $G \simeq G'$.

*Remark* 4. If $G = (V, A)$ and $G' = (V', A')$ are simple digraphs, i.e. digraphs without loops and multiple arcs, then $G \simeq G'$ if and only if there is a bijection $\phi : V \to V'$ such that $(u, v) \in A$ if and only if $(\phi(u), \phi(v)) \in A'$.

Since in our case the digraphs are simple, we use the definition given in remark 4.

Another notion we can recall is the **degree sequence** of a digraph. Since in a digraph $G$ every arc is oriented, for every vertex $v$ in $G$ is necessary to count how many arcs have tail in $v$ and how many have head in $v$. So every vertex $v$ has an **in-degree** $d_G^-(v)$ that is the number of arcs with head in $v$ and an **out-degree** $d_G^+(v)$ that is the number of arcs with tail in $v$. We simply call **degree** of a vertex $v$ the ordered pair $d_G(v) = (d_G^-(v), d_G^+(v))$; so the degree sequence of a digraph is the list of these pairs. With a notation borrowed from network theory, we call **source** a vertex with in-degree equal to zero and **sink** a vertex with out-degree equal to zero. All the other vertices are said to be **intermediate**.
The easiest property of isomorphic digraphs we can see is that they have the same degree sequence, that somehow means they have the same vertices.

**Proposition 2.1.1.** *Let $G = (V, A)$ and $G' = (V', A')$ be two digraphs.*
*If $G$ and $G'$ are isomorphic, then they have the same degree sequence.*

*Proof.* Let $u$ be a vertex of $G$.
Since $G \simeq G'$, there exists an isomorphism between these two digraphs; we call it $\phi$. So there is a vertex $v$ in $G'$ such that $\phi(u) = v$. From the definition of isomorphism we have that $\forall w \in V$ such that $(w, u) \in A$ there exists a vertex $w' \in V'$ such that $\phi(w) = w'$ and $(w', v) \in A'$.
$\Rightarrow d_G^-(u) = d_{G'}^-(v)$
Analogously for the out-degree:
$\forall w \in V$ such that $(u, w) \in A$ there exists a vertex $w' \in V'$ such that $\phi(w) = w'$ and $(v, w') \in A'$.
$\Rightarrow d_G^+(u) = d_{G'}^+(v)$
Therefore $u$ and $v$ have the same degree. Since this is valid for every vertex of G and every vertex has exactly one vertex in $G'$ associated with it by means of $\phi$, the two digraphs have the same degree sequence. □

Thanks to this proposition it is possible to see that if two digraphs don't have the same degree sequence, they cannot be isomorphic.
The second feature is about arcs. More precisely it states that two isomorphic digraphs have the same connections, that is they have arcs whose ending points have identical degrees.

**Proposition 2.1.2.** *Let $G = (V, A)$ and $G' = (V', A')$ be two isomorphic digraphs. Then there exists an injective relation between the two sets of arcs $f : A \to A'$ with the following property: if $e = (u, v)$ is an arc of $A$ and $f(e) = (u', v')$ is its image in $A'$ then $d_G(u) = d_{G'}(u')$ and $d_G(v) = d_{G'}(v')$.*

*Proof.* Let $\phi : V \to V'$ be the isomorphism between $G$ and $G'$.
As we have already proved in 2.1.1, $\forall v \in V$ we have $d_G(v) = d_{G'}(\phi(v))$.
Now let $e = (u, v)$ be an arc of $G$, that is $e \in A$. From remark 4, $e' = (\phi(u), \phi(v))$ is an arc of $G'$, that is $e' \in A'$. We can set $f(e) = (\phi(u), \phi(v))$ for every $e = (u, v) \in A$. This function satisfies the property required.
Moreover it is injective and well defined because $\phi$ is injective itself and well defined. It is quite easy to see this; for instance we can prove the injectivity since the well definition is similar.
Let $e = (u, v)$ and $a = (x, y)$ be arcs of $G$.
Suppose that $f(e) = f(a)$, i.e. $(\phi(u), \phi(v)) = (\phi(x), \phi(y))$. This means that $\phi(u) = \phi(x)$ and $\phi(v) = \phi(y)$. Since $\phi$ is a bijection, we have $u = x$ and $v = y$ that is $e = a$. $\square$

As in the case of proposition 2.1.1, we can use this second proposition to check whether two digraphs are not isomorphic. We simply need to find an arc $a = (u, v)$ in $G$ such that there isn't an arc $e = (x, y)$ in $G'$ with $d_G(u) = d_{G'}(x)$ and $d_G(v) = d_{G'}(y)$, so we cannot create the function $f$.
Checking the connections of the two digraphs leads us to a superficial knowledge of their structure. If we want to go deeper we can analyze all the possible paths of length two and then we can also study longer paths.
In order to keep the notation as clear as possible, we write the definition of directed path as it is given in [2].

**Definition 2.1.** A $(v_0; v_t)-$directed walk or a **directed walk** from $v_0$ to $v_t$ is an alternating sequence

$$W := (v_0, a_1, v_1, a_2, v_2, \dots, v_{t-1}, a_t, v_t)$$

of vertices and arcs where $a_i$ is an arc from $v_{i-1}$ to $v_i$ for all $i = 1, \dots, t$. Here, the vertices or arcs need not be distinct.

A $(v_0; v_t)-$path or a **path** from $v_0$ to $v_t$ is a $(v_0; v_t)-$directed walk in which all vertices and arcs are distinct.

From now on a path will be denoted with the sequence of its vertices because our digraphs are simple. Nevertheless the **length** of a path is the number of arcs it involves. The following proposition states that two isomorphic digraphs have the same paths of length two.

**Proposition 2.1.3.** *Let $G = (V, A)$ and $G' = (V', A')$ be two isomorphic digraphs. Then for all paths of length two $P = (u_1, u_2, u_3)$ with $u_i \in V\ \forall i = 1, 2, 3$ in $G$, there exists in $G'$ a path $P' = (v_1, v_2, v_3)$ of length two with $v_i \in V'\ \forall i = 1, 2, 3$ such that*

$$d_G(u_i) = d_{G'}(v_i)\ \forall i = 1, 2, 3.$$

*Proof.* Let $\phi : V \to V'$ be the isomorphism between $G$ and $G'$.
Every vertex $u_i$ in the path $P$ has an image $\phi(u_i)$ in $V'$ and as proved in 2.1.1 we have $d_G(u_i) = d_{G'}(\phi(u_i))\ \forall i = 1, 2, 3$.
Moreover $(u_1, u_2)$ is an arc from $u_1$ to $u_2$ in $G$, then $(\phi(u_1), \phi(u_2))$ has to be an arc from $\phi(u_1)$ to $\phi(u_2)$ in $G'$ because $\phi$ is an isomorphism. The same is valid for $(u_2, u_3) \in A$; $(\phi(u_2), \phi(u_3))$ is an arc in $G'$.
As a consequence we have that $(\phi(u_1), \phi(u_2), \phi(u_3))$ is a path from $\phi(u_1)$ to $\phi(u_2)$ with $d_G(u_i) = d_{G'}(\phi(u_i))\ \forall i = 1, 2, 3$.
Now we can denote $\phi(u_i) = v_i\ \forall i = 1, 2, 3$.
$\Rightarrow (v_1, v_2, v_3)$ is the searched path. □

The last property of isomorphic digraphs is an easy and natural extension of the previous one. It states that two isomorphic digraphs have the same paths of any length. Without loss of generality we can assume $k \leq |V|$ in the following proposition.

**Proposition 2.1.4.** *Let $G = (V, A)$ and $G' = (V', A')$ be two isomorphic digraphs. Then for all paths $P = (u_1, \dots, u_k)$ in $G$ with $u_i \in V\ \forall i = 1, \dots, k$, there exists in $G'$ a path $P' = (v_1, \dots, v_k)$ with $v_i \in V'\ \forall i = 1, \dots, k$ such that*

$$d_G(u_i) = d_{G'}(v_i)\ \forall i = 1, \dots, k.$$

*Proof.* If we name again $\phi$ the isomorphism between $G$ and $G'$, we can iterate the method used in 2.1.3.
In this way we obtain a path $P' = (v_1, \dots, v_k)$ in $G'$ where $v_i = \phi(u_i)$ $\forall i = 1, \dots, k$. Therefore $d_G(u_i) = d_{G'}(v_i)$ $\forall i = 1, \dots, k$ and $P'$ is the searched path. $\square$

## 2.2 Implementation of four tests

In this section we present four tests based on the previous propositions. These tests are written to check if two directed acyclic graphs are not isomorphic. They cannot give us a positive answer, so the word *True* here means "I don't know".
The general assumptions behind this implementation are: the analyzed graphs are acyclic, simple, their vertices are labeled with numbers from 0 to $n$ and there are no graphs with only sources and sinks.
The implementation of a graph is a list of pairs that represent its arcs. Even though the list is an ordered data type, here the order in which arcs are written is not relevant. For example the graph in 2.1 could be implemented with the list

$$[(0,1),(1,2),(1,3),(3,4),(3,5),(4,5),(5,2)]\,.$$

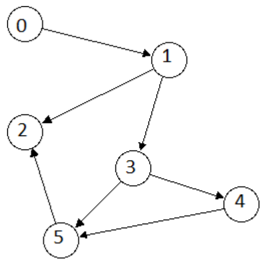

Figure 2.1: A directed acyclic graph.

In the following explanation the word "graph" refers to the list described above.

### 2.2.1 Degree sequences and edges

The first test we do on our graphs is based on proposition 2.1.1. So it returns *False* if the two degree sequences are different and this means that the two graphs

are not isomorphic, otherwise it returns *True*. To create the degree sequence is used *CreateBidegList* which takes a graph and returns a list $B$ of pairs. The element $B[i]$ is the degree of the vertex $i$ in the graph.
In order to check whether the two degree sequences are identical the function *CheckBideg* sorts a copy of them and uses the Python comparison operator ==.

**Example 2.1.** The degree sequence of the graph in 2.1 would be

$$[(0,1),(1,2),(2,0),(1,2),(1,1),(2,1)].$$

The second test is *CheckConnections* and it is based on proposition 2.1.2. First of all it invokes the function *EdgeBideg*, which takes a graph and its degree sequence as arguments. Then *EdgeBideg* creates a list with the arcs of the graph in which each vertex is replaced with its degree; below an example.

**Example 2.2.** Naming $G$ the list that implements the graph in figure 2.1 and $B$ its degree sequence we have

$$\begin{aligned}&\mathit{EdgeBideg}(G,B) = [((0,1),(1,2)),((1,2),(2,0)),((1,2),(1,2)),((1,2),(1,1)),\\&((1,2),(2,1)),((1,1),(2,1)),((2,1),(2,0))]\end{aligned}$$

Now the function *CheckConnections* is able to compare the two sets of arcs. It returns *True* if the two lists created by *EdgeBideg* are equal, *False* otherwise. In particular, for every element in the first list, the function looks for an identical element in the second one. If such an element is found, then it is deleted from the list, otherwise the function ends. This behavior reflects the injectivity of the function $f$ in proposition 2.1.2.

### 2.2.2 Paths of length two

At this point, if the previous tests gave positive response, we proceed to check the paths of length two with *CheckSimplePath*. The test can be summarize in three key points:

1. two dictionaries with the information of the graphs are created;
2. the function *AllSimplePath* creates all possible paths of length two in the first graph;

3. the function *FindSameSimplePath* searches in the second graph for the same paths generated in the previous step.

We can start analyzing the first point. An information dictionary for a directed graph has graph's vertices as keys and the values are lists such that the first element is the key's degree and the second one is the number of marks. These marks are needed in the third point because we want the function *FindSameSimplePath* to move in the direction less visited, so they indicate how many times a vertex has been used.

**Example 2.3.** At the beginning the dictionary of the graph in figure 2.1 would be:

$$D = \{0 : [(0, 1), 0], 1 : [(1, 2), 0], 2 : [(2, 0), 0], 3 : [(1, 2), 0], 4 : [(1, 1), 0], 5 : [(2, 1), 0]\}$$

In order to analyze the second point we need two more definitions; they are from [2].

**Definition 2.2.** Given a digraph $G = (V, A)$, the set

$$N_{out}(v) = \{x \in V | (v, x) \in A\}$$

is called the **set of out-neighbors** of $v \in V$, and the set

$$N_{in}(v) = \{x \in V | (x, v) \in A\}$$

is called the **set of in-neighbors** of $v \in V$.

The function *AllSimplePath* creates all paths of length two as follows.
For every intermediate vertex $v$ it creates the sets $N_{in}(v)$ and $N_{out}(v)$, then it matches each vertex in the first set with all vertices in the second one. In this way it is possible to have all paths of length two that have the vertex $v$ as central vertex.
Now, for every path of length two in the first graph, the function *CheckSimplePath* looks for an equal path in the second graph. To do this it invokes the function *FindSameSimplePath.* This last function has a path $p = (u_0, u_1, u_2)$ as argument and tries to create a path $sp = (v_0, v_1, v_2)$ in the second graph such that $d_G(u_i) = d_{G'}(v_i)$ $\forall i = 0, 1, 2$.
An important observation is that the response of this test depends on the order in which the graphs are entered as arguments, because it creates all possible paths on the first graph and looks for the same paths in the second graph. So changing the order in which the arguments are entered we can obtain a different response; we will see this with an example in the next chapter. For this reason the function *CheckSimplePath* is called twice when we compare two graphs.

### 2.2.3 Paths from a source to a sink

The last test we do is about complete paths, which are those that start with a source and end with a sink. Since our graphs are acyclic we always have such paths and following a path in the program we cannot create an infinite loop.
First of all the test *CheckLongPaths* creates all possible long paths in the first graph using the function *AllPathsInGraph*. For every source in the graph, this last function invokes *AllPathSource* to create all possible paths starting from that source. The algorithm used by *AllPathSource* is the following:

1. Given a source $s$, the set $N_{out}(s)$ is created.
2. For every vertex $u$ in $N_{out}(s)$ we do:
3. create a path $(s, u, u_1, u_2, \ldots, u_n)$ where $u_n$ is a sink and append it to a list *AllPaths*
4. take the last complete path $p = (s, w_1, \ldots, w_m)$ stored into *AllPaths*
5. starting with $t = m - 1$ we do:
   if the vertex $w_t$ is not exhausted, look for a vertex in $N_{out}(w_t)$ in order to create another complete path different from the previous ones, append the new complete path to *AllPaths* and go back to 4;
6. if $w_t$ is exhausted, repeat the step 5 with $t = t - 1$.
7. When the analyzed vertex is the source $s$, quit.

Here the word "exhausted" has the following meaning. Let $L = [p_1, \ldots, p_m]$ be the list of complete paths created so far. Suppose we are now analyzing the last path inserted $p_m = (u_0, \ldots, u_n)$. A vertex $u_t \in p_m$ is exhausted if for all $v \in N_{out}(u_t)$ there already exists a path in $L$ that contains $(u_{t-1}, u_t, v)$ as sub-path.
Now, fixed a long path $p$, *CheckLongPaths* looks for a similar path in the second graph using the function *SameLong*, which invokes *FindSameLongPathSource* for every source with same degree of the starting point of $p$.
Let $G = (V, A)$ and $G' = (V', A')$ be respectively the first and the second graphs given as arguments. Let $lp = (w_1, \ldots, w_m)$ be a path in $G$. The algorithm used by *FindSame-LongPathSource* is the following:

1. given a path $lp = [w_1, \dots, w_m]$ in $G$ and a starting point $s_1 \in V'$ with $d_{G'}(s_1) = d_G(w_1)$,
2. create a list for the searched path $sl = [s_1]$ and another for the no good directions, initially empty;
3. use an index $j$ to scan the path $lp$;
4. look for a vertex $v$ in the set of out-neighbors of the last element in $sl$ such that $v$ has the same degree of $lp[j]$ and $sl.append(v)$ is not a bad direction;
5. if you find it, append it to $sl$ and restart from 4
6. if you don't find it, the last vertex of $sl$ is a bad direction and it's not possible to proceed in the creation of $sl$: so append the $sl$ path to the list of bad directions and take off the elements of $sl$ until you find a vertex from which you can take a good direction;
7. if only bad directions are available and $sl$ has length 1, the searched path doesn't exist, so return error.

Here an elementary but clarifying example of how the algorithm works.

**Example 2.4.** Let our graphs be $F_1$ and $F_2$. They are isomorphic but they are almost regular, therefore since our criterion to distinguish vertices is their degree, we could say that they are all similar, except for the source and the sink. This will bring us a little more work to do.

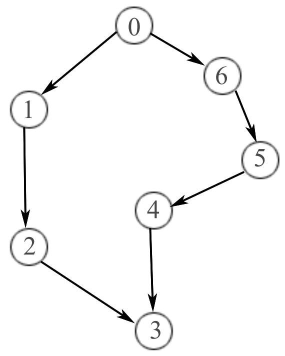

Figure 2.2: F1

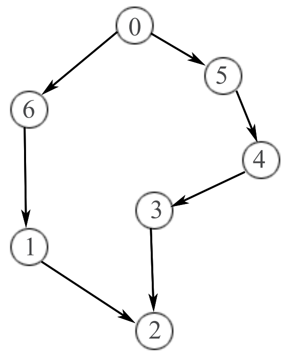

Figure 2.3: F2

We look for a path in $F_2$ similar to $lp = (0, 1, 2, 3)$ in $F_1$.
Since the source is unique we can start the path $sl$ with 0, so we have $sl = [0]$. Now we have two possibilities to proceed because $N_{out}(0) = \{5, 6\}$ and they have the same degree of $1 \in lp$ so they are both available. We choose 5 so $sl = [0, 5]$. Now the only possibility is 4 and it has the same degree of $2 \in lp$, so we add it to $sl$. At this point we should find a sink in the $N_{out}(4)$ because $3 \in lp$ is a sink in $F_1$, but there are no sinks in the neighbourhood of $4 \in F_2$. So this direction is a bad one and we need to take off 4 from $sl$. Then we look for a vertex in $N_{out}(5)$ different from 4 but there isn't. So we need to take off from $sl$ also 5 and look for another vertex in the neighbourhood of 0. This vertex is 6 and following the algorithm we don't find bad directions. Finally we obtain the path $sl = (0, 6, 1, 2)$ in $F_2$.

*Remark* 5. Note that the response of this test depends on the order we enter the arguments. So again we have to invoke it twice when we compare two graphs.

## 2.3 A finer partition

In this section we are going to analyze a partition of the vertices of a directed graph presented in [3]. In this paper the authors propose an algorithm to test the isomorphism of directed graphs. The algorithm tries to create a complete matching between the two

digraphs and it is based on the partition we report.
As we saw in 2.1.1, a digraph isomorphism preserves the degree of vertices. For this reason in the previous section, when we wrote "the same vertices" we were meaning "vertices with same degree". In fact the degree creates a partition of the vertices, but as we saw in the last example 2.4, sometimes this partition is very unrefined. In the partition given by the degree, a vertex is characterized by its relationship with all the adjacent vertices and nothing more. What seems to be more useful is a way to evaluate the relationship of a vertex with all other vertices in the graph.
This last information could be obtained from the distance matrix.

**Definition 2.3.** Given a graph $G = (V; E)$, the **distance matrix** D is an $|V| \times |V|$ matrix in which the element $d_{ij}$ represents the length of the shortest path from the vertex $v_i$ to $v_j$.

*Remark* 6.

- For every pair of vertices there is a unique minimum distance.
- If $i = j$ then $d_{ij}$ is zero.
- If a path doesn't exist between the two vertices, the length is defined to be infinite.

Since the distance matrix is a unique representation of a graph and it gives us information about the relationship among all the vertices in the graph, it can be used for our purpose.
We now give some notions that are needed to define the partition.

**Definition 2.4.** Given a digraph $G = (V, A)$ with $|V| = n$ we define a **row characteristic matrix** $XR$ to be an $n \times (n-1)$ matrix such that the element $xr_{im}$ is the number of vertices which are a distance $m$ away from $v_i$.
A **column characteristic matrix** $XC$ is an $n \times (n-1)$ matrix such that each element $xc_{im}$ is the number of vertices from which $v_i$ is a distance $m$.
A **characteristic matrix** $X$ is formed by the termwise juxtaposition of the corresponding rows of $XR$ and $XC$.

Given two digraphs $G^1 = (V, A)$ and $G^2 = (V', A')$, let $X^1 = (x^1_{lk})$ and $X^2 = (x^2_{lk})$ be the characteristic matrices of $G^1$ and $G^2$ respectively. An isomorphism will map $v^1_i \in V$

in $v_r^2 \in V'$ only if $x_{im}^1 = x_{rm}^2 \; \forall m$. Therefore vertices which exhibit identical rows of the characteristic matrix will be assigned to the same class. A **class vector** $C$ is a vector in which the element $c_i$ is the class of the vertex $v_i$.
The partition given by the characteristic matrix is often finer than the one given by the degree and it can never be less so. The following theorem is the proof of that.

**Theorem 2.3.1.** *If two vertices $v_i$ and $v_j$ are partitioned into separate classes by the degree, they will also be partitioned into separate classes by the characteristic matrix.*

*Proof.* For every vertex $v_i$, the element $xr_{i1}$ of the first column in $XR$ is $d_G^+(v_i)$ and the element $xc_{i1}$ of the first column in $XC$ is $d_G^-(v_i)$. This means that if two vertices have different degree, they will present a different element in the first column of the characteristic matrix. So they have different class. □

In order to give a criterion similar to that given by 2.1.1, we can define a **class count vector**, which is a vector $K$ such that the element $k_i$ is the number of vertices in class $i$. It follows that if $G$ is isomorphic to $G'$, then $k_i = k_i' \; \forall i$.
We now show an example in which the partition given by the distance matrix is much more refined than that given by the degree.

**Example 2.5.** We consider the directed acyclic graph in figure 2.4

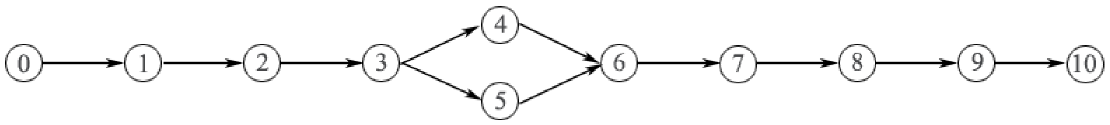

Figure 2.4: S1

The partition given by the degree is $\mathcal{P} = [(0), (1, 2, 4, 5, 7, 8, 9), (3), (6), (10)]$. So we have a block with seven elements.

The distance matrix of the graph $S1$ is

$$D = \begin{bmatrix} 0 & 1 & 2 & 3 & 4 & 4 & 5 & 6 & 7 & 8 & 9 \\ \infty & 0 & 1 & 2 & 3 & 3 & 4 & 5 & 6 & 7 & 8 \\ \infty & \infty & 0 & 1 & 2 & 2 & 3 & 4 & 5 & 6 & 7 \\ \infty & \infty & \infty & 0 & 1 & 1 & 2 & 3 & 4 & 5 & 6 \\ \infty & \infty & \infty & \infty & 0 & \infty & 1 & 2 & 3 & 4 & 5 \\ \infty & \infty & \infty & \infty & \infty & 0 & 1 & 2 & 3 & 4 & 5 \\ \infty & \infty & \infty & \infty & \infty & \infty & 0 & 1 & 2 & 3 & 4 \\ \infty & \infty & \infty & \infty & \infty & \infty & \infty & 0 & 1 & 2 & 3 \\ \infty & \infty & \infty & \infty & \infty & \infty & \infty & \infty & 0 & 1 & 2 \\ \infty & \infty & \infty & \infty & \infty & \infty & \infty & \infty & \infty & 0 & 1 \\ \infty & \infty & \infty & \infty & \infty & \infty & \infty & \infty & \infty & \infty & 0 \end{bmatrix}$$

Now it is easy to write the row characteristic matrix and the column characteristic matrix.
After calculating these two matrices it is possible to check that the characteristic matrix is the following.

$$X = \begin{bmatrix} 10 & 10 & 10 & 20 & 10 & 10 & 10 & 10 & 10 & 00 \\ 11 & 10 & 20 & 10 & 10 & 10 & 10 & 10 & 00 & 00 \\ 11 & 21 & 10 & 10 & 10 & 10 & 10 & 00 & 00 & 00 \\ 21 & 11 & 11 & 10 & 10 & 10 & 00 & 00 & 00 & 00 \\ 11 & 11 & 11 & 11 & 10 & 00 & 00 & 00 & 00 & 00 \\ 11 & 11 & 11 & 11 & 10 & 00 & 00 & 00 & 00 & 00 \\ 12 & 11 & 11 & 11 & 01 & 00 & 00 & 00 & 00 & 00 \\ 11 & 12 & 11 & 01 & 01 & 01 & 00 & 00 & 00 & 00 \\ 11 & 11 & 02 & 01 & 01 & 01 & 01 & 00 & 00 & 00 \\ 11 & 01 & 01 & 02 & 01 & 01 & 01 & 01 & 00 & 00 \\ 01 & 01 & 01 & 01 & 02 & 01 & 01 & 01 & 01 & 00 \end{bmatrix}$$

As a consequence the partition has only blocks of cardinality one except for a block of cardinality two. Looking at the rows of this matrix we can see that the fifth and the sixth row are identical, while the remaining rows are all different from each other. So the partition puts the vertex 4 in the same class of 5 -because our vertices are labeled starting from 0- and leaves all the others alone, each in a different class. In fact the class vector is $C = [1, 2, 3, 4, 5, 5, 6, 7, 8, 9, 10]$ and the partition is

$$\mathcal{P}' = [(0), (1), (2), (3), (4, 5), (6), (7), (8), (9), (10)].$$

A last observation that could be done is the following.
If we have graphs with few vertices, we might be tempted to try all possible permutations of the vertices' labels to check the isomorphism. The fact that the degree is preserved by an isomorphism allowed us to try not all the permutations but only those which associate vertices with same degree. Unfortunately this operation is sometimes very expensive. For instance if we want to check the isomorphism between the graph $S1$ and another one identical but with permuted labels, we should try 7! permutations, because we have seven vertices with same degree. Nevertheless, using the partition just presented we can reduce this number to 2, because we have a class with two elements and all the others with only one element.

# Chapter 3

# Examples

In this chapter we are going to try out our tests.
In particular in the first section we present some examples to explain how each test works and what kind of graphs it can distinguish.
The second section is dedicated to more regular acyclic graphs, that are the most difficult to analyze.

## 3.1 Some focused examples

### 3.1.1 Isomorphic graphs

As we described in the previous chapter, our tests are only able to say if two graphs are not isomorphic. This clearly means that if we use two isomorphic graphs, the response has to be positive.
Therefore the first pair of graphs we give as input to our tests is composed by the two isomorphic graphs in figure 3.1.

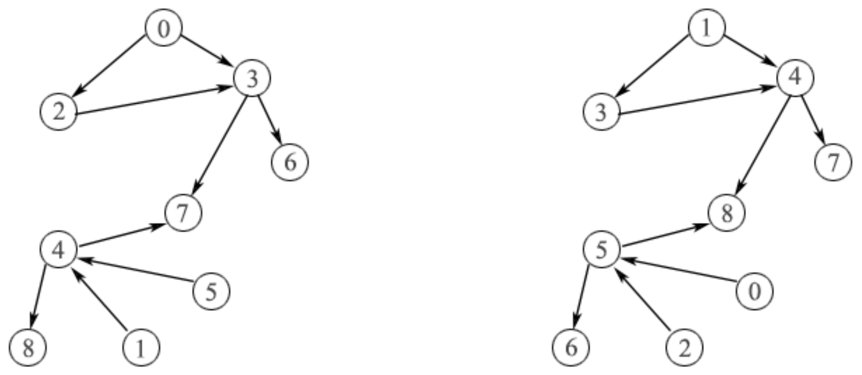

Figure 3.1: Two isomorphic directed acyclic graphs.

Using the tests with these graphs we obtain *True*.
If two graphs are isomorphic, then one is exactly the other but with permuted labels. For this reason, we simply generated a graph and then we permuted its labels using the *shuffle* function of Python. In order to check a huge number of graphs, we used a random generator of directed acyclic graphs.
We tested graphs with a number of nodes from five to twenty and for every graph we applied seven permutations. The response was positive in every test.

### 3.1.2 Graphs with different degree sequence and connections

The first test we do on our input graphs is based on proposition 2.1.1 while the second one is based on proposition 2.1.2. So we first check if the graphs have the same degree sequence and if the output is *True* we proceed with the second test.
In figure 3.2 we have three directed acyclic graphs. To simplify the explanation we call them $G = (V, A)$, $G_1 = (V_1, A_1)$ and $G_2 = (V_2, A_2)$.

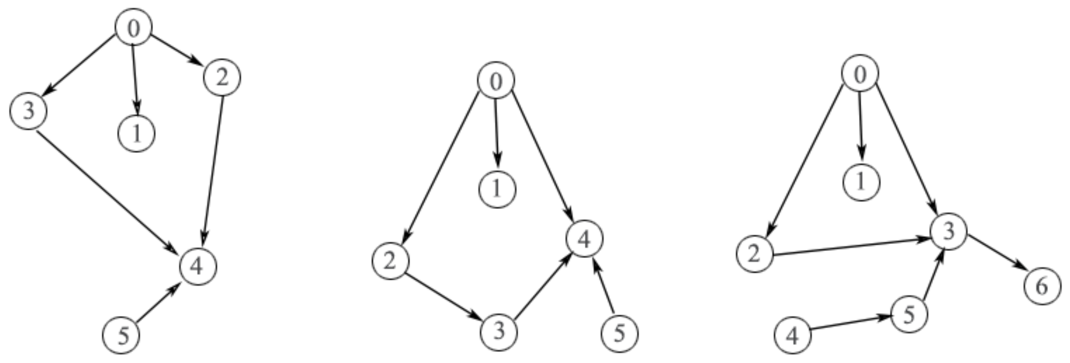

Figure 3.2: From the left: $G$, $G_1$, $G_2$.

If we invoke our tests on $G_1$ and $G_2$ we obtain that they have different degree sequence. Indeed the two degree sequences are respectively:

$$B_{G_1} = [(0,3),(1,0),(1,1),(1,1),(3,0),(0,1)]$$

$$B_{G_2} = [(0,3),(1,0),(1,1),(3,1),(0,1),(1,1),(1,0)]$$

Therefore they are not isomorphic and the program quits with output *False*.
The graph $G$ has instead the same degree sequence of $G_1$, so the response of the first test is *True*. Nevertheless they don't have similar connections because the arc $(2,3)$ in $G_1$ links two vertices with degree $(1,1)$ but in $G$ there isn't such an arc. This means that we cannot create a function $f : A_1 \to A$ as described in 2.1.2 because the arc $(2,3) \in A_1$ wouldn't have an image in $A$.
Obviously it is also impossible to create a function $f : A \to A_1$ as described in 2.1.2 because it should be injective. Indeed there are two different arcs in $A$ which link vertices with degree respectively $(0,3)$ and $(1,1)$; they are the arc $(0,3)$ and the arc $(0,2)$. But in $A_1$ there is only one arc which connects such vertices, that is $(0,2)$. So we might create the function $f$ but it wouldn't be injective, so it wouldn't be that in 2.1.2.

### 3.1.3 Graphs with different paths of length two

The third test is about paths of length two. As we mentioned in the previous chapter, since we create all possible paths of length two in the first graph given as input and then

we look for equal paths in the second graph, the response of this test depends on the order we enter the graphs. Now we give a clear example.
We consider the two graphs in figure 3.3 and we call them $G$ and $G'$. They have the same degree sequence and same connections, so the first two tests result *True* and we can proceed analyzing paths of length two.

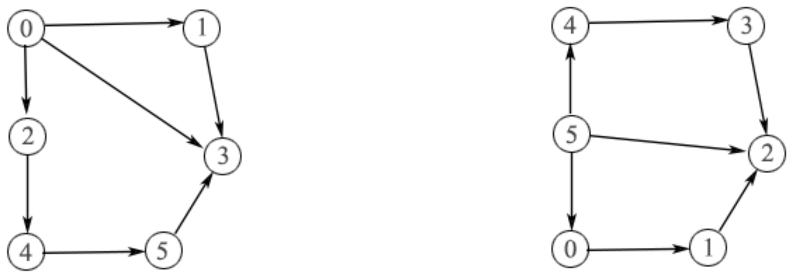

Figure 3.3: The graph $G$ on the left and the graph $G'$ on the right

The test which checks the paths of length two is *CheckSimplePath*. If we invoke this function on $(G', G)$, we obtain a positive response.
In fact the paths of length two in $G'$ are only of two different types. There are paths $(v_1, v_2, v_3)$ with

$$d(v_1) = (0,3),\ d(v_2) = (1,1),\ d(v_3) = (1,1) \tag{3.1}$$

or with

$$d(v_1) = (1,1),\ d(v_2) = (1,1),\ d(v_3) = (3,0). \tag{3.2}$$

From the figure 3.3 we can see that in $G$ we have both the paths' types. In particular, to the first set belongs $(0,2,4)$ while to the second $(4,5,6)$. So whenever we look for a path of the type 3.1 the program finds in $G$ the path $(0,2,4)$ and when we look for a path of the type 3.2 it finds $(4,5,6)$. This is the reason why, if we enter $G'$ as first argument and $G$ as second argument, the output is *True* even though the graphs are not isomorphic.
To make the test more strong and precise we decided to invoke it twice changing the order of the arguments. Indeed *CheckSimplePath*$(G, G')$ gives a negative response and makes the program end with the right output, that is *False*.
Analyzing the graph $G$, we find two additional types of paths $(v_1, v_2, v_3)$, that are

$$d(v_1) = (1,1),\ d(v_2) = (1,1),\ d(v_3) = (1,1) \tag{3.3}$$

and

$$d(v_1) = (0,3),\ d(v_2) = (1,1),\ d(v_3) = (3,0). \tag{3.4}$$

For instance, of the type 3.3 is $(2,4,5)$ and of the type 3.4 is $(0,1,3)$.
Since $G'$ has paths only of the type 3.1 and 3.2, it is impossible to find in it paths of the type 3.3 and 3.4.
Therefore using $G$ as first argument of *CheckSimplePath* and $G'$ as second one, we have output *False*.

### 3.1.4 Graphs with different complete paths

The last test we do to check if two graphs are not isomorphic, regards paths from a source to a sink. Again we invoke this test if and only if the previous have given positive response.
Two graphs that pass all the previous tests are $S1$, studied in the second chapter and shown in figure 2.4, and the graph $S2$ in figure 3.4.

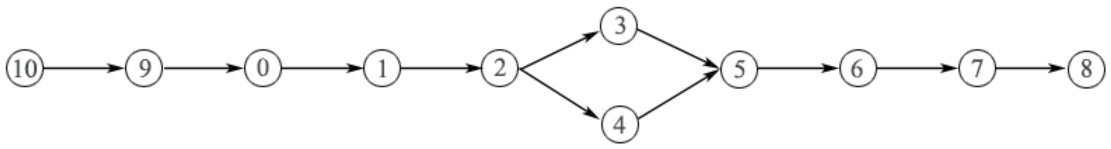

Figure 3.4: S2

These graphs have same degree sequences, same connections and same paths of length two, but it is easy to see that they don't have similar complete paths.
As we did in the previous example, we can divide the paths into types. Both graphs have only one type of paths. In $S1$ we have two complete paths of length 9, say $p = (v_1, \dots, v_{10})$, and the degrees of the vertices they involve are exactly

$$d(v_1) = (0,1), d(v_{10}) = (0,1), d(v_4) = (1,2), d(v_6) = (2,1), d(v_i) = (1,1) \forall i \in \{2,3,5,7,8,9\}.$$

On the other hand, $S2$ has again only two complete paths of length 9, say $p' = (v'_1 \dots, v'_{10})$, but the degrees are

$$d(v'_1) = (0,1), d(v'_{10}) = (0,1), d(v'_5) = (1,2), d(v'_7) = (2,1), d(v'_i) = (1,1) \forall i \in \{2,3,4,6,8,9\}.$$

To make the difference between the two paths types more evident we may substitute each vertex with its degree. So for the paths in $S1$ we have

$$((0,1);\ (1,1);\ (1,1);\ (1,2);\ (1,1);\ (2,1);\ (1,1);\ (1,1);\ (1,1);\ (0,1))$$

while for the paths in $S2$ we have

$$((0,1);\ (1,1);\ (1,1);\ (1,1);\ (1,2);\ (1,1);\ (2,1);\ (1,1);\ (1,1);\ (0,1))\,.$$

Therefore the graphs don't have the same paths from a source to a sink and they are not isomorphic.

### 3.1.5 Not isomorphic graphs with positive response

To conclude this section we present an example of two not isomorphic graphs for which our tests give positive response. This happens because there are some graphs more difficult to distinguish, they are those almost regular. Moreover, as we described in the second chapter, our tests are based on conditions that are only necessary for the isomorphism and not sufficient. So whenever our tests have output *True*, the analyzed graphs might be isomorphic or not; simply the program is not able to distinguish them. An example of this behavior might be observed using the graphs in the following figures. We call them $M$, in figure 3.5, and $N$, shown in 3.6. All our tests give a positive response on these graphs.

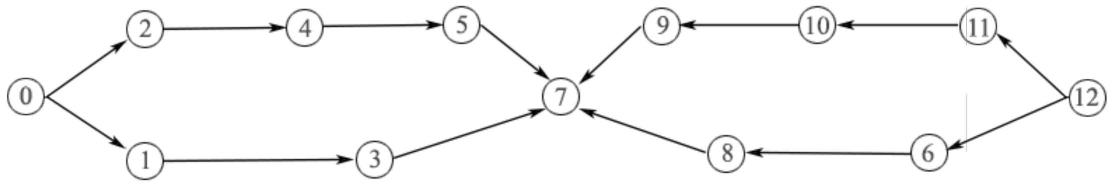

Figure 3.5: The graph $M$

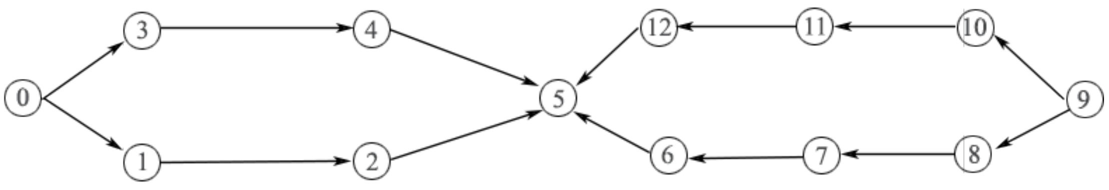

Figure 3.6: The graph $N$

These graphs are *almost* regular in the sense that their vertices have all the same degree except for the sources and the sinks. Therefore the partition of the vertices' sets $V$ and $V'$ given by the degree is very unrefined and the two graphs are difficult to distinguish.
Looking at the figures above, it is clear that the two graphs are not isomorphic. For instance one may use the chromatic number to establish this. In fact $N$ has chromatic number 2 while $M$ has 3.
Nevertheless, if we study only the paths, we can see they have the same paths both of length two and longer. In particular, all possible paths from a source to a sink in $M$ are

$$P_M = \{(0,1,3,7); (12,6,8,7); (0,2,4,5,7); (12,11,10,9,7)\} .$$

So when the program searches a path in $N$ similar to $(0,1,3,7) \in M$, it finds $(0,3,4,5)$ and again when it looks for a path similar to $(12,6,8,7) \in M$ it finds $(0,3,4,5) \in N$. The reason why the test makes these associations is that we asked it to check nothing but the degree of the vertices and these vertices have almost all the same degree.
Similarly for the paths of length four. The program finds the path $(9,10,11,12,5) \in N$ whether it looks for a path equal to $(0,2,4,5,7) \in M$ or to $(12,11,10,9,7) \in M$.

## 3.2 Acyclic regular graphs

As we saw with the last examples of the previous section and in the second chapter, there is a class of graphs more difficult to distinguish. They are the regular graphs.

In the undirected case, a graph $G = (V, E)$ is said to be k-regular if every vertex $v \in V$ has degree $k$. The extension of this definition to directed graphs is straightforward.
As we can read in [10], a directed graph $G = (V, A)$ is said to be **k-regular** if

$$d_G^+(v) = d_G^-(v) = k \ \forall v \in V.$$

So we don't simply need every vertex to have the same in-degree, or out-degree, of the others, but also these two numbers, in-degree and out-degree, need to be equal. The reason is that, in a directed graph, the following equality has to be valid:

$$\sum_{v \in V} d_G^+(v) = \sum_{v \in V} d_G^-(v).$$

Anyway our case is a bit different. Since our graphs are directed acyclic, we cannot have this kind of graphs. Indeed a necessary condition for a graph to be acyclic is to have at least one source and one sink. So we may force our graphs to have all vertices with same degree except for sources and sinks, i.e. all intermediate vertices have the same degree.
In order to check how much regular graphs are difficult to distinguish for our tests, we created a generator of acyclic regular graphs. Actually we don't generate directly such a graph, but we create an acyclic graph and then we "complete" it preserving the acyclic condition.
Therefore we start with a directed acyclic graph $G = (V, A)$ whose vertices are labeled with numbers from 0 to $n$ and $i < j$ for all arcs $(i, j) \in A$, because of the implementation of our random generator of directed acyclic graphs. Then we create a graph $G' = (V', A')$ which has $G$ as subgraph and $|V'| = n + 2d$, where $d$ is the maximum number of incoming or outgoing arcs for one single vertex $v \in V$. In particular, $d$ of these $2d$ more vertices we add, will play the role of sources in the graph $G'$ and the remaining $d$ will be sinks in $G'$. The reason why we need exactly this number of new sources and sinks is that we want every vertex of $G$ to have degree $(d, d)$ in $G'$ and $G'$ to be simple. Therefore a source in $G$ needs $d$ new sources in $G'$ to become an intermediate vertex with in-degree equal to $d$ as well as a sink in $G$ needs exactly $d$ new sinks in $G'$ to become a vertex with out-degree equal to $d$.
So, to create $G'$, we first calculate $d$ then we create $d$ new sources and $d$ new sinks. Now for every vertex $v \in V$ we add as many arcs as necessary to reach $d_{G'}(v) = (d, d)$. When

we have to add an arc, we may face two different situations. First, if we have to add an incoming arc to $v \in V$, we make it come from a source, i.e. we create an arrow $(s, v)$ where $s$ is one of the new $d$ sources contained in $V'$. Second, if we add an outgoing arc from $v \in V$, we choose to connect $v$ with the nearest vertex available in $G$, that means among all vertices $u \in V$ with $u > v$ and $d^-_{G'}(u) < d$ we choose the smallest. If no such vertex exists, for instance because all the vertices in $G$ bigger than $v$ already have in-degree equal to $d$, we connect $v$ with one of the new $d$ sinks contained in $V'$. Moreover, creating connections using always the nearest vertex less used, allows us to not create parallel edges and preserve the acyclic nature of the graph.
We now give an example of how the algorithm works.

**Example 3.1.** We consider the graph $G = (V, A)$ in figure 3.7 and we follow the algorithm step by step to make it become $G' = (V', A')$ in figure 3.8.

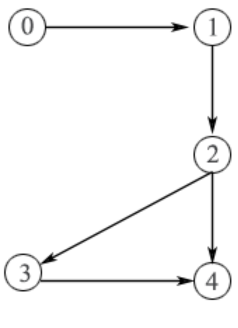
Figure 3.7: $G = (V, A)$

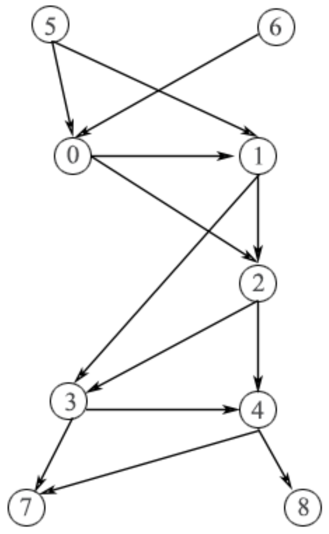
Figure 3.8: $G' = (V', A')$

In order to transform $G$ into $G'$ we have to calculate $d$. In this simple example we have $d = 2$ so we add to the graph four new vertices, everyone not connected with the others at the beginning. This passage is performed in figure 3.9.
Now we analyze each vertex of $G$ starting from 0. This vertex needs to be connected with two different vertices in order to make its indegree become 2. So we create two

incoming arcs, that are $(5, 0)$ and $(6, 0)$ as shown in figure 3.10.
To complete the process for 0 we add an outgoing arc connecting this vertex with the vertex 2. In figure 3.11 we see how the graph looks after these steps.

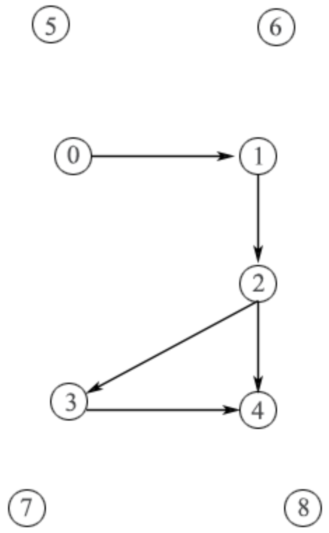

Figure 3.9: $G$ with two new sources and two new sinks

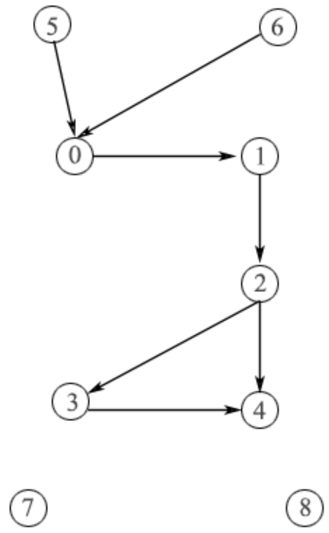

Figure 3.10: Addition of two incoming arcs to 0

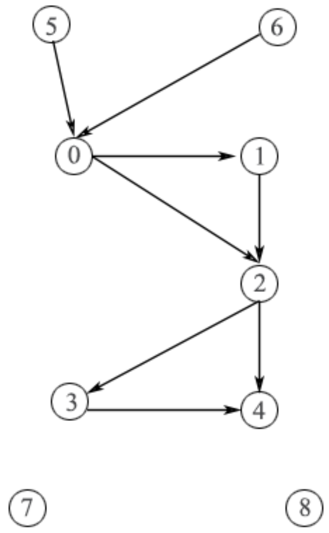

Figure 3.11: Addition of one outgoing arc from 0

Now we proceed with the vertex 1. It already has an incoming arc so we need to add only another one; we choose the source 5 to create this connection. Analogously for the outgoing arcs; the vertex 1 already has an outgoing arc so we add only one new arc $(1, 3)$. Notice that we choose the vertex 3 because 2 already has two incoming arcs, so we cannot add another one. This implies that 3 is the nearest available vertex for creating an outgoing arc from 1.
Figure 3.12 shows the situation after these steps.
Now we should complete 2 but it already has degree $(2, 2)$ so we can pass over it.
Analyzing 3 we see it has degree $(2, 1)$, so we need only to add an outgoing arc. We create $(3, 7)$ and we obtain the graph in figure 3.13.
To conclude the process we add two outgoing arcs to 4, that are $(4, 8)$ and $(4, 7)$. Now we have the almost regular graph $G'$.

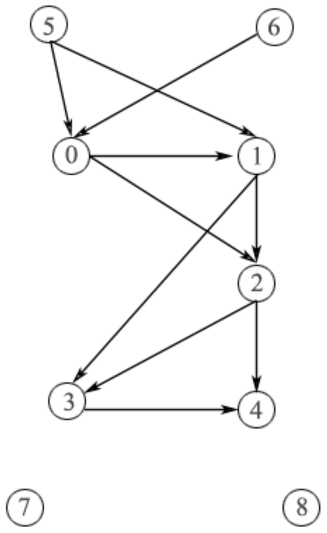

Figure 3.12: The graph after completing the vertex 1

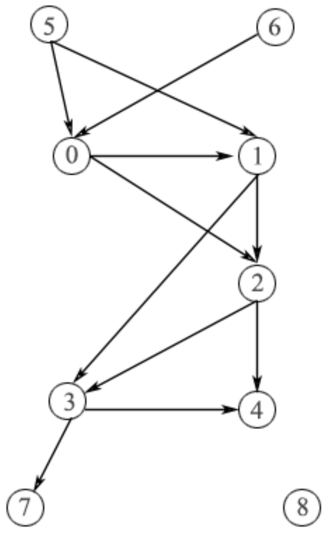

Figure 3.13: Addition of one outgoing arc from 3

To conclude the dissertation about acyclic regular graphs, we sum up in three tables the results obtained analyzing these graphs with our tests. In particular we want to check how many times our tests give positive response even though the graphs in input are not isomorphic.
First of all we generated 100 graphs in three different moments with $n = 5, 6, 7$ nodes, we made them almost regular and we divided them into classes with respect to the maximum number of incoming or outgoing arcs for a single vertex, i.e. the number so far indicated with $d$. Then in each class we paired the graphs, so we used, with every pair, both our tests and the function `is-isomorphic` of the library *Networkx*, described in the first chapter.
We tested the isomorphism only on graphs with same number $d$ because graphs with different $d$ are trivially not isomorphic and, since they have different degree sequences, our tests are able to distinguish them.
Moreover we excluded the two cases with $d = 1$ and $d = n - 1$, because those graphs are trivially isomorphic. In the case of $d = 1$, the graphs are "linear" and since they have the same number of vertices they are all isomorphic. In the second case, we have graphs that are "complete" in the sense that every vertex $i \in V$ has an incoming arc for

| Nodes | Degree | Graphs | Comparisons | Failures |
|---|---|---|---|---|
| 5 | $(2, 2)$ | 27 | 351 | 33 |
| 5 | $(3, 3)$ | 53 | 1378 | 217 |
| 6 | $(2, 2)$ | 7 | 21 | 6 |
| 6 | $(3, 3)$ | 40 | 780 | 56 |
| 6 | $(4, 4)$ | 37 | 666 | 232 |
| 7 | $(3, 3)$ | 19 | 171 | 12 |
| 7 | $(4, 4)$ | 45 | 990 | 180 |
| 7 | $(5, 5)$ | 29 | 406 | 226 |

Table 3.1: First attempt

all $j \in V$ with $j < i$ and an outgoing arc for all $k \in V$ with $k > i$.

In table 3.1 we sum up the results obtained in the first moment of testing. In the columns we have respectively the number of vertices in the basic graph $G$, the degree of the intermediate vertices in the new graph $G'$, the number of graphs generated, how many comparisons we did and how many times our response was *True* with not isomorphic inputs.

The first observation we could do is that, for fixed number of nodes in the basic graph, excluding the cases with too few comparisons, the number of failures grows with the degree, i.e. with the almost completeness of the graph. For instance the table 3.1 shows that in the case of degree $(4, 4)$ and 6 vertices of the basic graph, more than one third of the comparisons fails. And in the case with degree $(5, 5)$ and 7 vertices in the basic graph, about a half of the comparisons fails.

Another behavior we could observe, again excluding the cases with too few graphs, is that for fixed degree, the more the vertices in the basic graph are, the less our tests fail. This is the case of the degree $(3, 3)$: with 5 vertices in the basic graph we have a failure of about 15% but with 6 or 7 vertices in the basic graph the failures are about 7%.

The reason why this happens is that, if we have to create a fixed number of connections, the more vertices we have to choose among, the more is probably to create very different graphs.

| Nodes | Degree | Graphs | Comp. | Fail. |
|---|---|---|---|---|
| 5 | $(2,2)$ | 24 | 276 | 32 |
| 5 | $(3,3)$ | 53 | 1378 | 182 |
| 6 | $(2,2)$ | 8 | 28 | 5 |
| 6 | $(3,3)$ | 41 | 820 | 45 |
| 6 | $(4,4)$ | 33 | 528 | 98 |
| 7 | $(3,3)$ | 23 | 253 | 15 |
| 7 | $(4,4)$ | 37 | 666 | 85 |
| 7 | $(5,5)$ | 37 | 666 | 246 |

| Nodes | Degree | Graphs | Comp. | Fail. |
|---|---|---|---|---|
| 5 | $(2,2)$ | 29 | 406 | 38 |
| 5 | $(3,3)$ | 49 | 1176 | 144 |
| 6 | $(2,2)$ | 8 | 28 | 3 |
| 6 | $(3,3)$ | 37 | 666 | 30 |
| 6 | $(4,4)$ | 49 | 1176 | 490 |
| 7 | $(3,3)$ | 17 | 136 | 9 |
| 7 | $(4,4)$ | 46 | 1035 | 263 |
| 7 | $(5,5)$ | 25 | 300 | 118 |

Table 3.2: On the left the second attempt and on the right the third one

Here are two more tables with the results in other two different moments. Since the basic graphs are randomly generated it is not possible to observe precisely the two facts just described, anyway the global behavior reflects them.

# Chapter 4

# Future work

With this last chapter we want to introduce what might be the next step in the work just presented.
After studying the isomorphism problem, it could be interesting to analyze graphs from a different point of view. The main reason to do this is that, if two Morse graphs are not isomorphic, we could wonder how much they are effectively different and if it is however possible to say something about the multivector fields, from which they come, or not.
This different point of view is based on finite topological spaces. Therefore in the first section we present the theoretical background, while in the second section we describe its application to our context giving two examples.
The main reference for this chapter is [11] but some definitions are taken from [12], which sums up many concepts expressed in [11].
We omit the definition of homotopy groups but a complete treatise might be found in the fourth chapter of [13].

## 4.1 Finite topological spaces

A **finite topological space** is simply a topological space with a finite number of points. A topology on a finite set $X$ is a subset of the power set of $X$. For every point $x \in X$, the **minimal open set** $U_x$ is defined as the intersection of all open sets containing $x$. Minimal open sets constitute a basis for the topology of $X$, indeed any

open set $U$ of $X$ is a union of all $U_x$ with $x \in U$.
It is also possible to create a preorder on $X$ defining $x \leq y$ if $x \in U_y$.
On the other hand, if we have a preordered set $X$, we can define a topology on $X$. In particular we can choose as basis for our topology the family $\{y \in X | y \leq x\}_{x \in X}$. It is now easy to see that $y \leq x$ if and only if $y \in U_x$.
Therefore finite spaces and preoredered sets are basically the same objects considered from different perspectives. Moreover, in 1937 Alexandroff showed the correspondence between finite $T_0$ spaces and finite partially ordered sets. In fact, the antisymmetry of a partial order corresponds to the $T_0$ separation axiom. We remind that a topological space is said to be $T_0$ if for any pair of its points, there exists an open set containing one and only one of them.
Thanks to this correspondence we can move freely from the context of finite posets to the context of finite topological spaces and vice versa.
Here two consequences of this correspondence.

*Remark* 7. Open sets of finite spaces correspond to down-sets and closed sets to up-sets. A subset $U$ of a preordered set $X$ is a **down-set** if for every $x \in U$ and $y \leq x$, it holds that $y \in U$. The notion of up-sets is dually defined.

*Remark* 8. Given a finite topological space $X$ we have a preorder on it, so it is possible to speak of maximal, or minimal, element as well as of maximum, or minimum. In particular an element $x \in X$ is said to be **maximal** if $y \geq x$ implies $y = x$, and it is a **maximum** if $y \leq x$ for every $y \in X$.

### 4.1.1 Maps and homotopies

Since morphisms between topological spaces are continuous functions, we could wonder what a continuous function on finite topological spaces is. To answer this question we have the following result.

**Proposition 4.1.1.** *[11, Prop. 1.2.1] A function $f : X \to Y$ between finite spaces is continuous if and only if it is order preserving, i.e. $x \leq x'$ implies $f(x) \leq f(x')$ for every $x, x' \in X$.*

In order to give a description of homotopies in the context of finite topological spaces, we have to recall some notions of algebraic topology.

**Definition 4.1.** Let $X$ and $Y$ be topological spaces. The **mapping space** $Y^X$ denotes the set of maps from $X$ to $Y$. $Y^X$ can be considered a topological space by using the **compact-open topology**, i.e. the topology whose subbasis is given by the sets

$$W(C,U) = \left\{ f \in Y^X | f(C) \subseteq W \text{ for all } C \text{ compact in } X \text{ and } U \text{ open in } Y \right\}.$$

If $X$ and $Y$ are finite spaces we have that $Y^X$ is finite and we can consider on it the pointwise order, that is $f \leq g$ if $f(x) \leq g(x)$ for every $x \in X$.
An important remark is that every subspace of a finite space is compact because of the finiteness of the topology.
We recall now the definition of homotopic functions and homotopy equivalent spaces.

**Definition 4.2.** Two maps $f, g : X \to Y$ between topological spaces $X$ and $Y$ are **homotopic** if there exists a map, called **homotopy**, $G : X \times I \to Y$, where $I = [0,1]$, such that for all $x \in X$, $G(x;0) = f(x)$ and $G(x;1) = g(x)$. If two maps $f$ and $g$ are homotopic, we write $f \simeq g$.
Two spaces $X$ and $Y$ are said to be **homotopy equivalent** if there exist maps $f : X \to Y$ and $g : Y \to X$ such that $g \circ f \simeq id_X$ and $f \circ g \simeq id_Y$.
Given two maps $f, g : X \to Y$ such that for some $A \subset X$ we have $f|_A = g|_A$; an homotopy $G$ between $f$ and $g$ is said to be **relative to** $A$ if, $\forall t \in I$ we have $G|_{A \times \{t\}} = f|_A = g|_A$. In this case we write $f \simeq g$ rel $A$.

We also recall that a **path** between two points $x$ and $y$ of a topological space $X$ is a map $\alpha : I \to X$ such that $\alpha(0) = x$ and $\alpha(1) = y$.
If X is a finite space and $x, y \in X$ are two comparable points, then there exists a path from $x$ to $y$ (see [11, Lemma 1.2.3]). In fact, if we assume $x \leq y$, we can define $\alpha : I \to X, \alpha(t) = x$ if $0 \leq t < 1$, $\alpha(1) = y$. Now if $U \subseteq X$ is open and contains $y$, then it contains $x$ also. Therefore $\alpha^{-1}(U)$ is one of the sets $\emptyset, I$ or $[0,1)$, which are all open in $I$. Thus, $\alpha$ is a path from $x$ to $y$.
Moreover the existence of a path between to comparable elements $x, y \in X$ implies that there is a sequence of points $x_0, \ldots, x_n$ in X such that $x_0 = x$, $x_n = y$ and $x_i$ is comparable to $x_{i+1}$ for all $i = 1, \ldots, n-1$.

For finite spaces $X$ and $Y$, there is a natural correspondence between the set of homotopies $\{H : X \times I \to Y\}$ and the set of paths $\left\{\alpha : I \to Y^X\right\}$. The correspondence is simply given by $H(x;t) = \alpha(t)(x)$.
This fact implies that homotopy classes of maps between finite spaces are equivalent to path components in the mapping space and we have the following proposition.

**Proposition 4.1.2.** *[11, Corollary 1.2.6] Let $f, g : X \to Y$ be two maps between finite spaces. Then $f \simeq g$ if and only if there is a sequence of maps $f_0, \dots, f_n$ such that $f = f_0 \leq f_1 \geq f_2 \leq \dots f_n = g$.*

*Remark* 9. Any finite space X with maximum or minimum is contractible, i.e. it has the same homotopy type of a point. Indeed, in that case, the identity map $id_X$ is comparable with a constant map $c$, so thanks to 4.1.2 $id_X \simeq c$.

### 4.1.2 The classification theorem

When we are studying homotopy types of finite spaces, we can restrict ourselves to $T_0$-spaces. In fact, given a finite topological space $X$ we can create a particular quotient of it which is a $T_0$-space and has the same homotopy type of X.
The following proposition enables us to create such a quotient.

**Proposition 4.1.3.** *[11, Prop. 1.3.1] Let X be a finite space. Let $X_0$ be the quotient $X/\sim$ where $x \sim y$ if $x \leq y$ and $y \leq x$. Then $X_0$ is $T_0$ and the quotient map $q : X \to X_0$ is a homotopy equivalence.*

*Remark* 10. $X_0$ is homotopy equivalent to $X$.
In fact we can take the quotient map $q : X \to X_0$ and one of its sections $p : X_0 \to X$, which is by definition such that $q \circ p = id_{X_0}$. Thus $p \circ q$ is order preserving and furthermore, $p \circ q \leq id_X$ because it only sends elements to themselves or to something less than or equal to themselves. So using the proposition 4.1.2 we have $p \circ q \simeq id_X$. This means that $p$ is a homotopy inverse of $q$. So $X$ and $X_0$ have the same homotopy type.
Actually $X_0$ is a strong deformation retract of X, i.e. there exists a map $r : X \to X_0$ with $r|_{X_0} = id_{X_0}$ such that $i \circ r \simeq id_X$ rel $X_0$, where $i : A \to X$ is the inclusion map.

We give the definition of a particular type of points, whose removal doesn't affect the homotopy type of the space. First we remind that a point $x$ of a finite $T_0$-space $X$ **covers** another point $y \in X$ if in the Hasse diagram of the poset $X$ there is an arc $(x, y)$ from $x$ to $y$.

**Definition 4.3.** A point $x$ in a finite $T_0$-space X is a **down beat point** if $x$ covers one and only one element of X, or equivalently if the set $\hat{U_x} = U_x \setminus \{x\}$ has a maximum.
Dually, $x \in X$ is an **up beat point** if $x$ is covered by a unique element. Again this is equivalent to say that $\hat{F_x} = F_x \setminus \{x\}$ has a minimum, where $F_x = \{y \in X | y \geq x\}$ denotes the closure of $\{x\}$ in X.

In any of the two cases of the definition 4.3, we say that $x$ is a **beat point**. The following proposition shows the fundamental property of beat points.

**Proposition 4.1.4.** *[11, Prop. 1.3.4] Let $X$ be a finite $T_0$-space and $x \in X$ a beat point. Then $X \setminus \{x\}$ is a strong deformation retract of $X$.*

**Definition 4.4.** A finite $T_0$-space is a **minimal finite space** if it has no beat points.
A **core** of a finite space $X$ is a strong deformation retract which is a minimal finite space.

Thanks to propositions 4.1.4 and 4.1.3, we have that every finite space $X$ has a core and it is possible to find it by creating first the $T_0$-strong deformation retract $X_0 \subset X$, as shown in 4.1.3, and then removing beat points one by one till we obtain a minimal space.
The most important property of this process is given by the following theorem ([11, Corollary 1.3.7]).

**Theorem 4.1.5** (Classification theorem)**.** *A homotopy equivalence between minimal finite space is a homeomorphism. In particular the core of a finite space is unique up to homeomorphism and two finite spaces are homotopy equivalent if and only if they have homeomorphic cores.*

Therefore the core $X_c$ of a finite space $X$ is the smallest space homotopy equivalent to $X$. If $Y$ is another finite space homotopy equivalent to $X$, then the core of $Y$ must be homeomorphic to $X_c$.

### 4.1.3 Weak homotopy and simple homotopy type

Sometimes the spaces we have to compare do not have beat points. So we cannot use the method of reduction described above. In these particular cases we can however say something about our spaces using a tool a bit weaker than homotopy type, namely the weak homotopy type.
A map $f : X \to Y$ between topological spaces is said to be a **weak homotopy equivalence** if it induces isomorphisms in all homotopy groups, i.e. if $f_* : \pi_0(X) \to \pi_0(Y)$ is a bijection and the maps

$$f_* : \pi_n(X, x_0) \to \pi_n(Y, f(x_0))$$

are isomorphisms for every $n \geq 1$ and every base point $x_0 \in X$.
Homotopy equivalences are weak homotopy equivalences.

**Definition 4.5.** Two topological spaces $X$ and $Y$, not necessarily finite, are **weak homotopy equivalent**, or they are said to have the same weak homotopy type, if there exists a sequence of spaces $X = X_0, X_1, \ldots, X_n = Y$ such that there are weak homotopy equivalences $X_i \to X_{i+1}$ or $X_{i+1} \to X_i$ for every $0 \leq i \leq n-1$.

Note that, for finite spaces, weak homotopy equivalences are not in general homotopy equivalences and there exist weak homotopy equivalent spaces such that there is no weak homotopy equivalence between them. However if two spaces $X$ and $Y$ are weak homotopy equivalent, there always exists a third space $Z$ such that there exist weak homotopy equivalences $Z \to X$ and $Z \to Y$.
Even in this weak context there exists a notion similar to beat points: weak beat points.

**Definition 4.6.** Let X be a finite $T_0$-space. A point $x \in X$ is a **weak beat point**, or briefly a weak point, if either $\hat{U}_x$ is contractible or $\hat{F}_x$ is contractible. In the first case we say the $x$ is a **down weak point** while in the second case we say **up weak point**.

A useful observation is that beat points are in particular weak points thanks to remark 9. Moreover, for weak points, we have a result similar to proposition 4.1.4.

**Proposition 4.1.6.** *[11, Prop. 4.2.4] Let $x$ be a weak point of a finite $T_0$-space $X$. Then the inclusion map $i : X \setminus \{x\} \hookrightarrow X$ is a weak homotopy equivalence.*

Thanks to this proposition it is possible to remove weak points from a space $X$ to obtain another space with the same weak homotopy type and it is contained in $X$. Unfortunately, when we compare spaces, we rarely have one contained into the other, so we may not use the proposition 4.1.6. In these cases we probably need not only to remove points but also to add them, so as to create a sequence of weak homotopy equivalences and then show the two spaces have the same weak homotopy type.
The following definitions are necessary to present the solution to this problem.

**Definition 4.7.** Let X be a finite $T_0$-space and let $Y \subset X$. We say that X collapses to Y by an **elementary collapse** (or that Y expands to X by an elementary expansion) if Y is obtained from X by removing a weak point. We denote $X \overset{e}{\searrow} Y$ (or $Y \overset{e}{\nearrow} X$).
In general, given two finite $T_0$-spaces X and Y, we say that X **collapses** to Y (or Y expands to X) if there is a sequence $X = X_0, X_1, \dots, X_n = Y$ of finite $T_0$-spaces such that for each $0 \leq i < n$, $X_i \overset{e}{\searrow} X_{i+1}$. In this case we write $X \searrow Y$ (or $Y \nearrow X$).
Two finite $T_0$-spaces X and Y are **simple homotopy equivalent** if there is a sequence $X = X_0, X_1, \dots, X_n = Y$ of finite $T_0$-spaces such that for each $0 \leq i < n$, $X_i \searrow X_{i+1}$ or $X_i \nearrow X_{i+1}$.

It is possible to prove that homotopy equivalent finite $T_0$-spaces are simple homotopy equivalent.
Therefore, given two finite $T_0$-spaces X and Y, we have the following sequence of implications:

X and Y are homotopy equivalent

$$\Downarrow$$

X and Y are simple homotopy equivalent

$$\Downarrow$$

X and Y are weak homotopy equivalent

This means that if we want to transform a space into another without changing its weak homotopy type, we can do it by simply removing or adding weak points.

To conclude, if we have two finite topological spaces $X$ and $Y$ and we want to compare their homotopy types, we may proceed as follows. We can create two finite $T_0$-spaces $X_0$ and $Y_0$ homotopy equivalent to $X$ and $Y$ respectively. Removing beat points from $X_0$ and $Y_0$ we obtain the cores $X_c$ of $X$ and $Y_c$ of $Y$. If the two cores are homeomorphic, then $X_0$ and $Y_0$ are homotopy equivalent and so are also $X$ and $Y$. If the two cores are not homeomorphic, $X_0$ and $Y_0$ don't have the same homotopy type but they could be weak homotopy equivalent. To see this we may use proposition 4.1.6, if the spaces are cointained one into the other, or the strategy of collapses. Since homotopy equivalences are particular kind of weak homotopy equivalences, if $X_0$ is weak homotopy equivalent to $Y_0$, so are $X$ and $Y$.

## 4.2 Application to multivector fields

The theory of finite topological spaces might be suitably applied to multivector fields. In fact, we start with a finite Lefschetz complex, which is itself a finite topological space, and define on it a dynamics by means of a multivector field. As we saw in section 1.1.3, every multivector field $\mathcal{V}$ uniquely defines a directed graph $G_{\mathcal{V}}$. Moreover we defined a relation $\prec_{\mathcal{V}}$ on the Lefschetz complex $X$ given by the arrows of $G_{\mathcal{V}}$. This relation induces a preorder $\leq_{\mathcal{V}}$ on $X$. Therefore the preordered set $(X, \leq_{\mathcal{V}})$ corresponds to a finite topological space which doesn't satisfy the $T_0$ separation axiom. In order to obtain a $T_0$-space, we may use the proposition 4.1.3. The resulting $T_0$-space is that given by the Morse graph of $\mathcal{V}$. In fact, the equivalence relation given in 4.1.3 is exactly our decomposition in strongly connected components.
Therefore our directed acyclic graph $G$ with the relation $u \leq v$ if there is an arc in $G$ from $u$ to $v$, corresponds to a finite $T_0$-space with the topology of down, or up, sets. It is straightforward that we can now use the theory described in the previous section to compare this kind of graphs.
If the two directed acyclic graphs we are analyzing result not isomorphic, we may ask how much they are different. So we can study their homotopy type, and weak homotopy type, to draw some conclusions on the multivector fields, using the classification theorem 4.1.5.

Unfortunately, this approach might sometimes give very scarce information. For instance, all the graphs examined in the third chapter are contractible, so if they were Morse graphs, they would come from spaces which are homotopy equivalent one another.
Moreover the study of weak homotopy type might sometimes induce us to consider as similar graphs which actually look very different. We show this fact in two examples. In the first example we use two graphs that are not isomorphic but look very similar because one can be obtained from the other by removing a vertex. And in the second example we show that two graphs, which don't look similar, have the same weak homotopy type.

**Example 4.1.** We consider the two directed acyclic graphs in figure 4.1. We call them $R$ and $R1$. They are clearly not isomorphic, because $R$ has a vertex less than $R_1$. Moreover they cannot have homeomorphic cores because $R_1$ doesn't have beat points, so its core is itself. Therefore the two graphs don't have the same homotopy type, but we are going to prove they are weak homotopy equivalent.

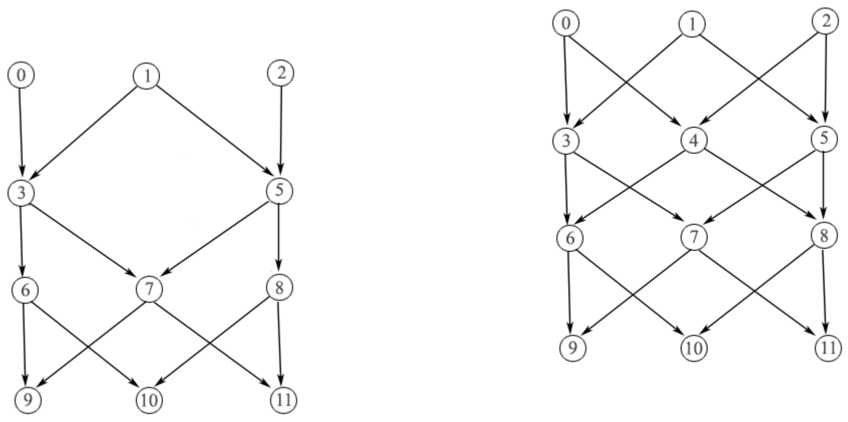

Figure 4.1: On the left the graph $R$ and on the right the graph $R_1$

From the figure it is easy to notice that we may obtain $R$ from $R_1$ by removing the vertex 4. Therefore we start analyzing that vertex; it is an up-weak point for $R1$. To see this we have to consider $\hat{F}_4$ in $R1$, which is shown in figure 4.2. This subspace is contractible. Indeed the vertices 9 and 11 are two down-beat points because both $\hat{U}_9$ and

$\hat{U}_{11}$ are composed of a single vertex. Therefore we can remove from $\hat{F}_4$ these two vertices obtaining a subspace with a maximum, which is contractible because of remark 9. This proves that 4 is a weak point of $R_1$ and, thanks to 4.1.6, we can remove it preserving the weak homotopy type. So $R$ and $R_1$ are weak homotopy equivalent.

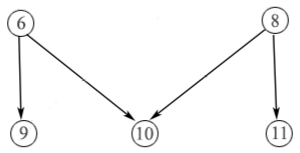

Figure 4.2: $\hat{F}_4$ in $R1$.

**Example 4.2.** Now we compare the graph $R_1$, shown in figure 4.1, with the graph $R_2$ in figure 4.3. Again they are not isomorphic and don't have homeomorphic cores. In fact, the core of $R_1$ is $R_1$ itself, while $R_2$ is contractible.

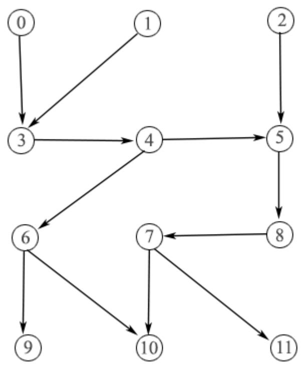

Figure 4.3: The graph $R_2$.

To see that $R_2$ is contractible we may simply notice that the vertices 2 and 1 are both up-beat points. Indeed the set $\hat{F}_2 = \{5, 7, 8, 11, 10\}$ has a minimum, which is 5, and the set $\hat{F}_1 = \{3, 4, 5, 6, 7, 8, 9, 10, 11\}$ has a minimum too, which is 3.

Thanks to proposition 4.1.4 we can remove these two vertices without modifying the homotopy type of $R_2$. As shown in figure 4.4, the new graph $R_2^1$ has a minimum, which is the vertex 0. Therefore it is contractible, because of remark 9, and its core is a single point.

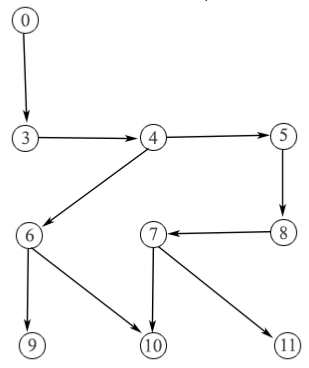

Figure 4.4: The graph $R_2^1$.

We can conclude that $R_2$ and $R_1$ don't have homeomorphic cores so they are not homotopy equivalent. Nevertheless they could have the same weak homotopy type but they are not contained one into the other. So, to show they are weak homotopy equivalent, we have to use the strategy of elementary collapses and expansions.
As we have already shown, the graph $R_2$ can be reduced to a single vertex, removing beat points, that are in particular weak points. Therefore we start with this vertex to reconstruct $R_1$. The single vertex which is the core of $R_2$ becomes the vertex 0 of $R_1$ and then we add in order: vertices 3 and 4 that are down-beat points, the 1 and 2 that are up-beat points. So we obtain the graph $R_1^1$ in figure 4.5.

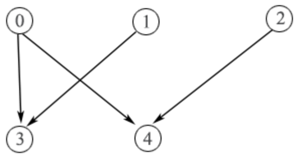

Figure 4.5: Graph $R_1^1$

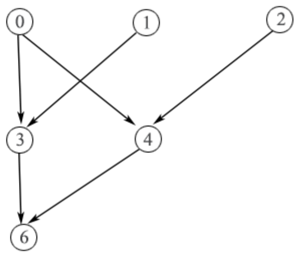

Figure 4.6: Graph $R_1^2$

Now we can add the vertex 6 which is a down-weak point. In fact $\hat{U}_6$ is $R_1^1$ which is contractible. So we obtain the graph $R_1^2$ in figure 4.6.
To this last graph we add the vertices 7 and 10, which are both down-beat points, obtaining the graph $R_1^3$.
Now we add to $R_1^3$ the vertex 5, which is up-beat, creating the graph $R_1^4$.
Then we add the vertex 8 which is again up-beat, and we obtain the graph $R_1^5$.

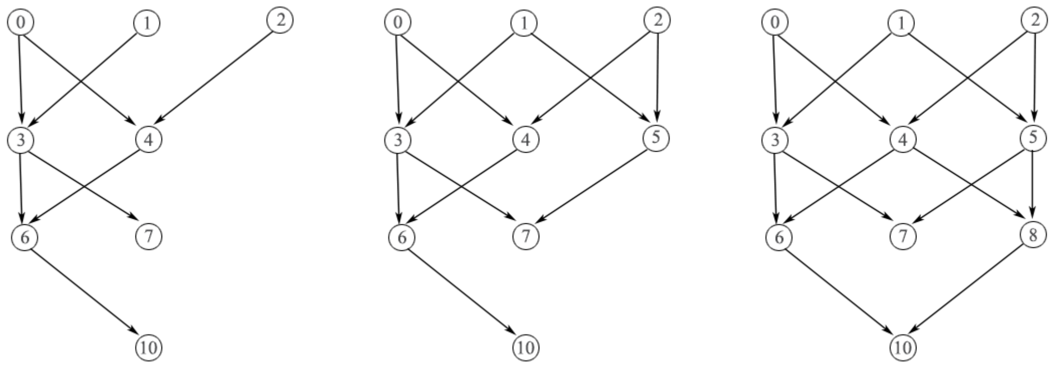

Figure 4.7: From the left: $R_1^3$, $R_1^4$, $R_1^5$.

The last step to obtain $R_1$ consists in adding to $R_1^5$ the vertices 9 and 11 which are two weak points. In this way we have the following sequence of collapses and expansions, where $\bullet$ represents the graph with a single vertex:

$$R_2 \searrow R_2^1 \searrow \bullet \nearrow R_1^1 \nearrow R_1^2 \nearrow R_1^3 \nearrow R_1^4 \nearrow R_1^5 \nearrow R_1.$$

As a consequence we have that $R_1$ and $R_2$ are simple homotopy equivalent and so they are weak homotopy equivalent.

# Appendix A

# The code

Here we present the code used in our implementation. It is divided in four modules. The module presented in the first section contains the four tests describe in section 2.2 and the function which invokes them sequentially.
The module PATHS contains functions for the creation and manipulation of paths in a directed acyclic graph. It is presented in section A.2.
Two basics modules are GRAPH, which contains functions for working directly on the directed acyclic graph, and a second module with auxiliary functions. They are in sections A.3 and A.4 respectively.
In section A.5 we have two scripts: one to generate a directed acyclic graph and the other to make a directed acyclic graph become more regular, as we described in paragraph 3.2.

*Remark* 11. We remind that we worked with the general assumption that our graphs were acyclic, simple and with only few vertices. Therefore our implementation cannot be used to analyze graphs which have cycles and its performance, in terms of time, depends on the number of vertices and edges. Moreover this code is based on a representation of the directed acyclic graph as that we described in section 2.2.

## A.1 The module for testing the isomorphism

```
from PATHS import*

def CheckBideg(B1,B2):
    #input: two sets of arcs
    #output:true if the degree sequences are equal,
    #        false otherwise
    Bideg1=sorted(B1)
    Bideg2=sorted(B2)
    return(Bideg1==Bideg2)

def CheckConnections(G1,G2,BidegA,BidegB):
    #input:two graphs G1 and G2, two degree sequnces
    #output:true if the graphs have the same connections,
    #        false otherwise
    E1=EdgeBideg(G1,BidegA)
    E2=EdgeBideg(G2,BidegB)
    for e in E1:
        if e not in E2:
            return(False)
        else:
            i=index(E2,e)
            del E2[i]
    return(True)

def CheckSimplePath(A,B,dictA,dictB,Inter):
    #input: two graphs A and B, two info dictionaries,
    #        list Inter of intermediate vertices in A
    #output:True B has the same short paths of A,
    #        False otherwise
    infoA=dictA.copy()
    infoB=dictB.copy()
    Paths=AllSimplePath(infoA,A,Inter)
    error=False
    i=0
    while i<len(Paths) and not error:
        p=Paths[i]
```

```
            same=FindSameSimplePath(p,infoA,infoB,B)
            if same==[]:
                error=True
            else:
                for u in same:
                    infoB[u][1]=infoB[u][1]+1
            i=i+1
        return(not error)

    def CheckLongPaths(P,B, dictA, dictB):
        #input: list P of all paths in A, the graph B,
        #        two info dictionaries
        #output:True if B has the same long paths of A
        infoA=dictA.copy()
        infoB=dictB.copy()
        error=False
        j=0
        while not error and j<len(P):
            path=P[j]
            res=SameLong(path,infoA,infoB,B)
            if not res:
                error=True
            j=j+1
        return(not error)

    def CompleteTest(A,B):
        #input: two lists of arcs
        #output:False if they are not isomorphic,
        #        True otherwise
        BidegA=CreateBidegList(A)
        BidegB=CreateBidegList(B)
        B1=BidegA.copy()
        B2=BidegB.copy()
        f=CheckBideg(B1,B2)
        if f==False:
            return(False,'Different bidegree lists')
        s=CheckConnections(A,B,BidegA,BidegB)
        if s==False:
```

```
        return(False, 'Different connections')
    infoA=CreateInfoDict(A)
    infoB=CreateInfoDict(B)
    Inter=ListInter(infoA)
    if Inter==[]:
        return(False, 'only sinks and sources')
    else:
        t=CheckSimplePath(A,B,infoA,infoB,Inter)
        if t==False:
           return(False, 'second graph has different short paths')
        else:
            interB=ListInter(infoB)
            #if interA!=[], then interB!=[]
            t=CheckSimplePath(B,A,infoB,infoA,interB)
            if t==False:
                return(False,'first graph has different short paths')
    dictA=infoA.copy()
    AllPathsA=AllPathsInGraph(A,dictA)
    l=CheckLongPaths(AllPathsA,B, infoA, infoB)
    if l==False:
        return(False,'second has different long paths')
    dictB=infoB.copy()
    AllPathsB=AllPathsInGraph(B,dictB)
    l1=CheckLongPaths(AllPathsB, A, infoB, infoA)
    if l1==False:
        return(False,'first has different long paths')
    return(True,"I don't know")
```

## A.2 The module PATHS

```
from GRAPH import*

def FindSameSimplePath(path,D1,D2,B):
    #input: path (a,b,c) in the graph A, two info dictionaries D1 and D2,
    #        the set of arcs of the graph B
    #output:(x,y,z) the path found in the second graph,
    #        found=[] it this path doesn't exist
    a=path[0]
```

```
        b=path[1]
        c=path[2]
        poss=[]
        same=[]
        for e in D2:
            if D2[e][0]==D1[b][0]:
                poss.append(e)
        i=0
        found=False
        while i<len(poss) and not found:
            #Create the star of edges
            y=poss[i]
            N=BackwardNeig(y,B)
            #look for the first element of same
            x=BestChoice(a,N,D1,D2)
            F=neigbourhood(y,B)
            z=BestChoice(c,F,D1,D2)
            if x==-1 or z==-1:
                i=i+1
            else:
                same.append(x)
                same.append(y)
                same.append(z)
                found=True
        return(same)

    def AllSimplePath(D,A,Inter):
        #input: info dictionary D, set of arcs A,
        #       set of intermediate vertices Inter
        #output: list of all paths of length two
        AllPath=[]
        for b in Inter:
            FN=neigbourhood(b, A)
            BN=BackwardNeig(b,A)
            for a in BN:
                for c in FN:
                    path=[a,b,c]
                    AllPath.append(path)
    return(AllPath)

def LongPath(s,A,path,AP):
    #input: starting point s, the set of arcs A, a path, the list
    #       of all paths just created using s as starting point
    #output: a complete path
    N=neigbourhood(s,A)
    if N==[]:#s is a sink
        return(path)
    else:
        found=False
        i=0
        while not found and i<len(N):
            j=index(path,s)
            sublist=[path[j-1],s,N[i]]
            if not is_contained(sublist,AP):
                found=True
            i=i+1
        path.append(N[i-1])
        return(LongPath(N[i-1],A,path,AP))

def AllPathSource(s,A,D):
    #input: a source s, the set of arcs A, the info dictionary D
    #output: a list with all complete paths from the source s
    N=neigbourhood(s,A)
    AllPath=[]
    for x in N:
        AllPath_x=[]
        p=LongPath(x,A,[s,x],AllPath_x)
        AllPath.append(p)
        AllPath_x.append(p)
        last_path=AllPath_x[len(AllPath_x)-1]
        pointer=len(last_path)-1
        while last_path[pointer]!=s:
            b=last_path[pointer]
            a=last_path[pointer-1]
            if not is_exhausted(a,b,A,AllPath_x) and not is_sink(b,D):
                k=index(last_path,b)
```

```
                    path=last_path[0:k+1]
                    new_path=LongPath(b,A,path,AllPath_x)
                    AllPath.append(new_path)
                    AllPath_x.append(new_path)
                    last_path=new_path
                    pointer=len(last_path)-1
                else:
                    pointer=pointer-1
        return(AllPath)

    def AllPathsInGraph(A,D):
        #input: set of arcs A, info dictionary D
        #output: all the complete paths in A
        Res=[]
        Sources=ListSources(D)
        for s in Sources:
            L=AllPathSource(s,A,D)
            for p in L: #Union of lists
                Res.append(p)
        return(Res)

    def FindSameLongPathSource(s1,lp,D1,D2,B):
        #input: a source s1 in the graph B, a complete path lp in the graph A,
        #        the two info dictionaries D1 and D2, the set of acrs of B
        #output: a complete path in B similar to lp if it exists,
        #        otherwise an empty list
        i=1
        error=False
        NoGood=[]
        samelong=[s1]
        v=s1
        j=0
        while i<len(lp) and not error:
            u=lp[i]
            N=neigbourhood(samelong[j],B)
            poss=[]
            for x in N:
                tmp=samelong.copy()
            tmp.append(x)
            if tmp not in NoGood:
                poss.append(x)
        v=BestChoice(u,poss,D1,D2)
        if v==-1:
            if i==1:
                error=True
                samelong=[]
            else:
                NoGood.append(samelong)
                samelong=samelong[0:len(samelong)-1]
                i=i-1
                j=j-1
        else:
            samelong.append(v)
            j=j+1
            i=i+1
            D2[v][1]=D2[v][1]+1
    return(samelong)

def SameLong(lp,D1,D2,B):
    #input: a complete path in A, two info dictionaries,
    #        the set of arcs of B
    #output:True if there exists in B a path similar to lp,
    #        False otherwise
    S=ListSources(D2)
    found=False
    i=0
    while not found and i<len(S):
        if D2[S[i]][0]==D1[lp[0]][0]:
            slp=FindSameLongPathSource(S[i],lp,D1,D2,B)
            if slp!=[]:
                found=True
        i=i+1
    return(found)
```

## A.3 The module GRAPH

```
from AUX_FUNCTIONS import *

def CreateBidegList(A):
    #input: a set of arcs
    #output: its degree sequence
    vertex=0
    Bideg=[]
    while vertex<NumVertex(A):
        outdeg=0
        indeg=0
        for e in A:
            if e[0]==vertex:
                outdeg=outdeg+1
            if e[1]==vertex:
                indeg=indeg+1
        vertex=vertex+1
        Bideg.append([indeg,outdeg])
    return(Bideg)

def CreateInfoDict(A):
    #input: list of arcs
    #output:info dictionary
    #       {vertex:[(indeg, outdeg), marks]}
    InfoDict=dict()
    vertex=0
    while vertex<NumVertex(A):
        outdeg=0
        indeg=0
        for e in A:
            if e[0]==vertex:
                outdeg=outdeg+1
            if e[1]==vertex:
                indeg=indeg+1
        InfoDict[vertex]=[(indeg, outdeg),0]
        vertex=vertex+1
    return(InfoDict)
```

```
def ListSources(D):
    #input: a info dictionary
    #output: list of sources sorted with respect to degree
    L=VertexSorted(D)
    S=[]
    i=0
    while L[i][1][0][0]==0:
        S.append(L[i][0])
        i=i+1
    return(S)

def ListInter(D):
    #input: a info dictionary
    #output:list of intermediate vertices
    L=[]
    for e in D:
        if D[e][0][0]!=0 and D[e][0][1]!=0:
            L.append(e)
    return(L)

def is_sink(v,D):
    #input: a vertex v in a graph G,
    #        the info dictionary of G
    #output:True if v is a sink of G
    #        False otherwise
    return(D[v][0][1]==0)

def is_source(v,D):
    return(D[v][0][0]==0)

def EdgeBideg(A,Bideg):
    #input: A set of arcs, D info dictionary of A
    #output:list of arcs with degree instead of vertices
    E=[]
    for e in A:
        a=e[0]
        b=e[1]
```

```
            E.append((Bideg[a],Bideg[b]))
        return(E)

    def neigbourhood(u, A):
        #input: u vertex of a graph A
        #output:the set of out-neighbors
        Neig=[]
        for e in A:
            if e[0]==u and e[1] not in Neig:
                Neig.append(e[1])
        return(Neig)

    def BackwardNeig(u,A):
        #input: u vertex of a graph A
        #output:the set of in-neighbors
        Back=[]
        for e in A:
            if e[1]==u and e[0] not in Back:
                Back.append(e[0])
        return(Back)

    def BestChoice(u,N,D1,D2):
        #input:D1 info dictionary of graph G1, D2 info dict of graph G2
        #       u vertex of G1, N subset of vertices in G2
        #output:a vertex v in N with same degree of u and
        #       minumum number of marks if it exists,
        #       -1 otherwise
        found=False
        i=0
        res=-1
        poss=[]
        while i<len(N):
            x=N[i]
            if D2[x][0]==D1[u][0]:
                poss.append(x)
            i=i+1
        if poss!=[]:
            res=MinMarks(poss,D2)
    return(res)

def is_exhausted(a,b,A,AllPaths):
    #input: a,b vertices of A,
    #       AllPaths list of paths in A
    #output:True is the vertex b is exhausted
    #       False otherwise
    F=neigbourhood(b,A)
    ShortPaths=[]
    for f in F:
        ShortPaths.append([a,b,f])
    exhausted=True
    i=0
    while exhausted and i<len(ShortPaths):
        p=ShortPaths[i]
        if not is_contained(p,AllPaths):
            exhausted=False
        i=i+1
    return(exhausted)


def ChooseSourceSink(L, minim): #for regular acyclic
    #input: L list of degrees of sources or sinks, minim a vertex
    #output:a source or a sink with minimun number of connections
    v=minim
    deg=max(L[0][0],L[0][1])
    for j in range(1,len(L)):
        if max(L[j][0],L[j][1])<deg:
            v=minim+j
            deg=max(L[j][0],L[j][1])
    return(v)

def NotAlreadyUsed(G,v): #for acyclic generator
    #input: a graph G and a vertex v
    #output:-1 if there is an edge in G with v as ending point
    #otherwise it returns a number between 0 and v-1
    i=0
    while i<len(G):
            if G[i][1]==v:
                return(-1)
            i=i+1
        return(random.randint(0,v-1))

    def ChooseNearestAvailable(L,d, minim, G, w): #for acyclic regular
        #input: L list of degrees of vertices, d degree,
        #       minim minimun vertex available,
        #       G graph, w vertex to connect
        #output: the nearest vertex available if it exists,
        #       -1 otherwise
        v=-1
        i=0
        found=False
        while not found and i<len(L):
            if L[i][0]<d and (w,i+minim) not in G :
                v=minim+i
                found=True
            i=i+1
        return(v)
```

## A.4 The auxiliary functions

```
    import time

    def NumVertex(A):
        #input: list of arcs
        #output: number of vertices
        maxim=0
        for e in A:
            if e[0]>maxim:
                maxim=e[0]
            if e[1]>maxim:
                maxim=e[1]
        return(maxim+1)

    def Choice(t):
        #Auxiliary function used in VertexSorted
    return(t[1][0])

def VertexSorted(D):
    L=sorted(D.items(), key=Choice)
    return(L)

def index(L,e):
    #input: L list, e in L
    #output:index of e in L
    i=0
    while i<len(L):
        if L[i]==e:
            return (i)
        i=i+1
    return(-1)

def SameList(U,Used):
    #Checks if two lists are equal
    for e in U:
        if e not in Used:
            return(False)
    return(True)

def MinMarks(N,D):
    #input: N list of verteces of G, D info dictionary of G
    #output: a vertex in N with minimum number of marks
    res=N[0]
    mark=D[res][1]
    for e in N:
        if D[e][1]<mark:
            res=e
            mark=D[e][1]
    return(res)

def is_contained(subList,L):
    #input: two lists
    #output:True if in L there is a element that contains subList
    #       False otherwise
```

```
        listindex=0
        found=False
        while not found and listindex<len(L):
            p=L[listindex]
            j=1
            while j<len(p)-1:
                if p[j-1]==subList[0] and p[j]==subList[1] and p[j+1]==subList[2]:
                    found=True
                j=j+1
            listindex=listindex+1
        return(found)

    def use_permutation(Edges, sigma):
        #input: a list Edges of pairs (x,y), a permutation sigma
        #output: a list of pairs (sigma(x),sigma(y))
        Res=[]
        for e in Edges:
            newedge=(sigma[e[0]],sigma[e[1]])
            Res.append(newedge)
        return(Res)
```

## A.5 Two scripts to generate directed acyclic graphs

We assume that the Python library *random* and the module GRAPH are imported. The following code generates a directed acyclic graph. The inputs are two positive numbers. The former is the number of nodes we want in the graph and the latter expresses the probability to create an edge between two vertices.

```
    def my_gen(num_nodes,percent):
        graph=[]
        Used=[]
        j=0
        while j<num_nodes-1:
            for k in range(j+1, num_nodes):
                tmp=random.randint(0,100)
                if(tmp<percent):
                    graph.append((j,k))
                    if j not in Used:
```

```
                    Used.append(j)
            if j not in Used:
                node=random.randint(j+1,num_nodes-1)
                graph.append((j,node))
            j=j+1
        node=NotAlreadyUsed(graph, num_nodes-1)
        if node!=-1:
            graph.append((node,num_nodes-1))
        return(graph)
```

With the following script it is possible to transform a directed acyclic graph, created with the generator just presented, into a regular acyclic graph.

```
    def regular_acyclic(G):
        n=NumVertex(G)
        V=CreateBidegList(G)
        d=0
        R=G.copy()
        for e in V:
            if d<max(e[0], e[1]):
                d=max(e[0],e[1])
        for k in range(0, 2*d):
            V.append([0,0]) #add d sources and d sinks
        i=0
        while i<n:
            if V[i][0]<d: #addition of incoming arcs
                tot=d-V[i][0]
                for j in range(0,tot):
                    sources=V[n:n+d]
                    s=ChooseSourceSink(sources, n)
                    R.append((s,i))
                    V[i][0]=V[i][0]+1
                    V[s][1]=V[s][1]+1
            if V[i][1]<d: #addition of outgoing arcs
                tot1=d-V[i][1]
                for j in range(0,tot1):
                    poss=V[i+1:n]
                    u=ChooseNearestAvailable(poss,d, i+1, R, i)
                    if u==-1: #no intermediate vertices are available
                            sinks=V[n+d:] #so choose a sink
                            u=ChooseSourceSink(sinks, n+d)
                        R.append((i,u))
                        V[i][1]=V[i][1]+1
                        V[u][0]=V[u][0]+1
                i=i+1
            return (R,d)
```

# Bibliography

[1] J.A. Bondy and U.S.R. Murty. Graphs Theory with Applications. North-Holland, 1976.

[2] S.A. Choudum. Graph Theory, module 11. A NPTEL Course.

[3] D. C. Schmidt and L. E. Druffel. A fast backtracking algorithm to test directed graphs for isomorphism using distance matrices. Journal of the Association for Computing Machinery, Vol 23 No 3 July 1976 pp 433 − 445.

[4] M. Mrozek, Conley-Morse-Forman theory for combinatorial multivector fields on Lefschetz complex. Foundations of computational mathematics (2016)

[5] T. Kaczynski, K. Mischaikow, M. Mrozek. Computational Homology, Applied Mathematical Sciences 157, Springer-Verlag, 2004.

[6] T. K. Dey, M. Juda, T. Kapela, J. Kubica, M. Lipinski, M. Mrozek. Persistent Homology of Morse Decompositions in Combinatorial Dynamics.

[7] L. P. Cordella, P. Foggia, C. Sansone, M. Vento. An Improved Algorithm for Matching Large Graphs. 3rd IAPR-TC15 Workshop on Graph-based Representations in Pattern Recognition, Cuen (2001)

[8] L. P. Cordella, P. Foggia, C. Sansone, M. Vento. Performance evaluation of the VF graph matching algorithm. Proceedings. International Conference on Image Analysis and Processing, 1999.

[9] A. Hagberg, D.Schult, P.Swart. NetworkX Reference. Release 2.0.

[10] Chen, Wai-Kai (1997). Graph Theory and its Engineering Applications. World Scientific.

[11] J.A.Barmak. Algebraic topology of finite topological spaces and applications. Lecture notes in Mathematics, Springer. (2011)

[12] J. Wakefield. Finite spaces and applications to the Euler characteristic.

[13] A. Hatcher. Algebraic topology. Cambridge University Press.(2002)

[14] H. Kokubu, K. Spendlove, A User's Guide to the Conley-Morse Database.

[15] M. Mrozek, B. Batko, Coreduction homology algorithm. Discrete and Computational Geometry, 41(2009), 96-118.

[16] A. V. Aho, J. E. Hopcroft, J. D. Ullman. The design and analysis of computer algorithms. Addison-Wesley Publishing Company, 1974.

# Ringraziamenti

Il primo, grande, ringraziamento va sicuramente al professor Massimo Ferri che mi ha dato l'opportunità di vivere questa meravigliosa esperienza a Cracovia. Lo ringrazio soprattutto per il suo supporto, i suoi incoraggiamenti ed i suoi insegnamenti.
Vorrei ringraziare anche il prof. Mrozek, per avermi aiutata con la burocrazia e accolta non appena arrivata, ed il dott. Juda, per avermi guidata nello studio e nella preparazione di questa tesi.
Per quanto riguarda la mia esperienza a Cracovia, posso affermare con certezza che il ringraziamento più grande va a tutti i meravigliosi e fortissimi membri della SGI Poland con cui ho condiviso questi bellissimi mesi. Grazie di cuore a tutti i membri della divisione giovani che mi hanno accolta alla loro prima riunione generale il 7 Ottobre 2017. Grazie al fortissimo gruppo Ovest di Cracovia; i loro zadankai sono stati davvero delle oasi di pace e serenità per me. Grazie ad Ania, Ewa, Marcin, Kunal, Ruchi, Justyna, Mitsuko, Mariko, Agata e tutti gli altri. Grazie davvero per tutto quello che mi avete insegnato e per tutta la forza che mi avete dato in questi mesi. Vi porterò sempre nel cuore.
Grazie a Sara insostituibile compagna di grandi risate.
E grazie a Daniela, per avermi trovata subito e non avermi lasciata più. Ci siamo divertite un sacco insieme e anche da lei ho imparato tanto. Sono certa che avremo ancora molte esperienze da condividere. E non importerà in quali parti del mondo ci ritroveremo a vivere.
Prima della mia partenza sono stati fondamentali per me anche molti membri della SGI Italy, che vorrei ringraziare. Un super grazie al fortissimo gruppo Maestrale di Bologna, che mi ha accompagnata e sostenuta negli ultimi due anni. Con loro e grazie a loro è iniziata la mia rivoluzione umana. Non li dimenticherò mai. Grazie a tutti i meravigliosi

compagni della divisione giovani che mi hanno sempre incoraggiata. Grazie anche a tutto il gruppo Vittoria di Ancona, che mi ha sempre accolta a braccia aperte ogni volta che ero di ritorno ad Ancona. Grazie alle bellissime giovani donne con cui anche durante queste ultime vacanze natalizie ho condiviso stupendi momenti di studio e confronto. Sono contenta di tornare da loro.

Con questa tesi si conclude un lungo percorso di studio che ho condiviso pienamente con due persone speciali: Elena e Ilaria. Meravigliose compagne di avventure e amiche sincere. Anche da loro ho imparato molto. Non le ho mai viste vacillare, nemmeno un secondo. Le ho viste vivere e superare momenti difficilissimi, soprattutto negli ultimi anni, senza mai fare un singolo passo indietro. Sono davvero fortissime e gli voglio un bene immenso.

Grazie a Marta, per il suo sostegno e le belle chiacchierate.

Grazie a tutti i miei amici. In particolare Sara, Martina, Luca e Matteo, che hanno sempre creduto in me. Ci siamo sempre divertiti tanto insieme e sono contenta di poter dire che questi cinque anni, in cui ho vissuto lontana da loro, non hanno minimamente intaccato il nostro legame di amicizia. È proprio vero che certe persone ce le portiamo a fianco vita dopo vita.

L'ultimo gigante, enorme, immenso, infinito ringraziamento va a tutta la mia grande e meravigliosa famiglia. Grazie a zia Lucy, zio Rena, zio Franco e zia Irene. Ognuno di loro ha partecipato a questo mio percorso. Ognuno di loro mi è stato vicino a modo suo in questi anni e lo ringrazio con tutto il cuore. Grazie alle mie dolcissime nonne, Paola e Sita. Anche loro compagne di grandi chiacchierate ed inestinguibili fonti di saggi consigli. Grazie a nonno Gianni, che fino all'ultimo ha continuato a chiedermi come andasse lo studio.

Grazie a Linda per le risate e le chiacchierate al telefono. Sono fierissima di lei e sono certa che realizzerà grandi cose.

Grazie ad Andrea, che mi supporta e sopporta ormai da molto tempo, e spero abbia voglia di farlo ancora. In questi mesi mi sono resa conto che siamo in grado di vivere lontani, in città diverse, ma ho anche notato che le mie giornate, senza avere lui vicino, sono un po' più grigie, un po' meno allegre. Mi è mancato e non vedo l'ora di poter passare di nuovo il tempo con lui.

Grazie, grazie, grazie, mille volte grazie a zia Cry e zio Serge, che mi hanno accompagnata in questa avventura. Si sa che quando uno studente si trasferisce all'estero per un periodo di studio, i primi giorni sono i più difficili. Si dice che ci sia una vera e propria crisi. Io in quei giorni avevo loro e quindi non ho vissuto nessuna crisi, ma solo tante risate. In questa occasione, come in tantissime altre, non avrei saputo che fare senza di loro. Sono davvero una delle più grandi certezze che ho nella vita.
So che non mi lasceranno mai sola.
Infine un ringraziamento speciale alla mia famiglia quella piccola, ma piccola solo per questioni di numeri. La famiglia dentro la famiglia.
Grazie a mamma, papà e Gabriele.
Non c'è molto da dire. Se ci sono loro, io mi sento a casa. Se ci sono loro è come se ci fossero anche tutti gli altri. Con loro mi sono sempre fatta le risate migliori, dico sul serio, i discorsi più profondi e le discussioni più sincere. Grazie a tutti e tre per le occasioni che mi avete sempre dato. Soprattutto grazie per tutte le volte che mi avete fatto crescere. Grazie per tutti i confronti che ho avuto e avrò con voi. Perchè li ho sempre sentiti -e li sentirò- arrivare davvero infondo al cuore per illuminarne un pezzettino. Grazie perchè se c'è un pochino di luce in me, ce l'avete messa voi. Grazie perchè se sono così felice è solo grazie a voi. Voi tre. Mamma, papà e Gabriele. Grazie. Vi voglio bene.
A tutti, vi voglio bene!