Università degli Studi di Camerino

Scuola di Scienze e Tecnologie

Corso di Laurea in Informatica (Classe L-31)

PERSISTENT HOMOLOGY APPLICATION FOR FINDING ALL ELEMENTARY CYCLES IN A GRAPH

Laureando
Lorenzo Rossi
Matricola 087467

Relatore
Prof.ssa Emanuela Merelli

Correlatore
Dott. Matteo Rucco

Secondo Correlatore
Jacopo Binchi

A.A. 2013/2014

Abstract

L'obiettivo di questa tesi è di proporre un nuovo approccio per la risoluzione del problema della ricerca di tutti cicli minimi in un grafo. Il problema della ricerca dei cicli rigurda la teoria dei grafi. Ad oggi sono stati pubblicati alcuni algoritmi in grado di risolvere il problema eseguendo una visita del grafo per ogni ciclo presente. Anche se questi algoritmi risolvono con successo il problema, la loro complessità computazionale risulta più che polinomiale: infatti il problema della ricerca dei cicli appartiene alla classe dei problemi NP-completi.
Noi presentiamo un nuovo approccio al problema, l'omologia persistente. L'omologia è un meccanismo per ricercare cicli in ogni dimensione. L'omologia persistente deriva dall'omologia algebrica e permette di caratterizzare forme (nel senso geometrico del termine) da un dato spazio. Noi mostriamo come l'applicazione di questa teoria può individuare con successo tutti i cicli minimali in un grafo. In conclusone, comparando il nostro algoritmo con quelli classici, possiamo concludere che l'omologia persistente riduce la complessità computazionale che, nel caso peggiore, è $O(n^3)$

The aim of my thesis is to introduce a new approach for face-up the problem of finding all elementary cycles in a graph. The cycle detection problem is a problem regarding graph theory, at the best of our knowledge only few algorithms able to solve that problem was published. All of this procedures have a solution based on the visit of the graph, one for each cycle. Even if these algorithms successfully solve the problem, their computational complexity is more than polynomial: cycle detection belongs to the class of NP-complete problems.
We present a new approach of looking at the problem, the persistent homology. Homology is a machinery for finding loop in each dimension. Persistent homology is the grand child of algebraic homology and it allows to characterize shapes (in geometrical sense) from a space. We show how the application of this theory can successfully detect all minimal cycle in a graph. In conclusion we compare our algorithm with the classic ones: we can conclude that persistent homology reduces the complexity of the problem, in the worst case to $O(n^3)$.

Contents

1. Introduction

A lot of situations in mathematics, science and also in the real world can be described by means of a diagram consisting of a set of points together with lines joining some pairs of these points. For instance a politician that is talking to the public could be represented as a point connected with each the spectator. A graph gives that mathematical abstraction of a situation: in fact a graph is a set of point that could be in relation with another point in the set. The points are called vertex or nodes, while the relations are called edges, links or arcs. In graph theory there exist two main families of graphs: directed and undirected graphs. In a undirected graph an edge from a to b do not have a direction so a and b are connected each other. Instead in a directed graph an edge from a to b is directed, so b is not connected to a. Another subset of graphs are the weighted ones. A graph is weighted if we associate a weight to each edge. A lot of problems can be solved using graph theory and some of these have a fast resolution. Another part of graph theory problems belong to the *NP-complete* class of problems: a problem is *NP-complete* if it requires a time more than polynomial to be solved. For instance a practical usage of weighted graphs is the *Travelling Salesman Problem*[12] that is *NP-complete*. We are interested in a problem about graph theory: find all elementary cycles in a graph. Think about to take a journey in your country: it is easy to imagine cities as a set of vertices and roads like the edges. Every road and city we passed describe a path. Once we come back to home the path ends at the start point: this path describe a cycle. If we visit each cities once we describe an elementary cycle. This problem has two different prospective depending on the kind of graph: in a directed graph we have to note that an edge is like a one way road and a cycle can not pass it in the wrong direction. In the last fourty years only a few algorithms were presented to solve the cycle detection problem. We considered a group of these algorithms that share a similar idea: a graph has a cycle if a *DFS* visit meets the start vertex twice. There are various ways to traverse of a graph. One of these ways is the *DFS*, depth-first search, that gives us some information about graph structure. In depth-first search the idea is to travel as deep as possible from neighbour to neighbour before backtracking and follow another path. What determines how deep is possible to go is that we must follow edges, and we can not visit any vertex twice. With a soft alteration of this method we could verify if during the visit some edge go back to the start node. In this thesis we show a new solution to the problem. We use persistent homology to find all elementary cycles in a graph. Topology is the branch of geometry that studies shapes, and its main feature is that it classifies objects according to some properties that persist under some possible transformations. As we explain in the relative chapter, persistent homology find holes in a graph. We can decide which to says witch hole is described by an elementary cycle and consequently we can solve the cycle detection problem.
In the following chapters we propose an introduction to the graph theory and compu-

tational topology where we list the formal definitions useful to understand the domain. We chose some reference algorithms that, at the state of the art, are the fastest. We describe how they work and we analyze their computational complexity. In order to have a comparison we implemented all these procedures in *Java* using a few libraries useful to our work: *JHoles*[2] and *jGraphT*[1] to manipulate graphs, *javaplex*[9] to calculate the persistent homology. Finally we compare the running times of the implementations in order to have touchable result.

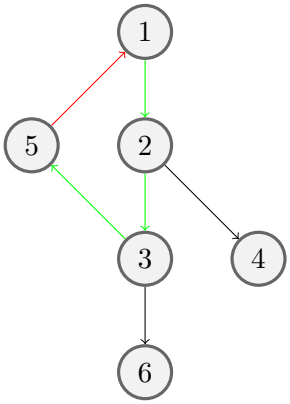

Figure 1.1: In green the path described by a DFS visit, the red edge close a cycle

2. Graph Theory

Basically a graph represent a set of objects (nodes, vertices): their may be connected by a relation (edge, link). In computer science, a graph has several usage: it may describe the topology of all routers connections in a network, relationships among users of a social network and so on. Formally graphs are mathematical instruments, so they have a precise definition.

2.1 Graph

Definition 2.1.1 (Graph)
A graph G(V,E) is a ordered pair of finite and (not empty) sets (V,E). V contains all the nodes and E contains edges that are pairs of nodes.

Definition 2.1.2 (Edge)
An edge e in a graph G(V,E) is a binary relation $\varepsilon(v,w)$ such that $v,w \in V$

Definition 2.1.3 (Complete graph)
A graph G(V,E) is complete if $\forall v, w \in V \exists \varepsilon(v, w) \in E$.

2.1.1 Measures

Definition 2.1.4 (Order or Volume of a graph)
Given a graph G(V,E), the order (or rank) of G, $|G|$, is the number of element of V.

Definition 2.1.5 (Size of a graph)
Given a graph G(V,E), the size of G, $||G||$, is the number of the element of E.

Definition 2.1.6 (Density of a graph)
Given a graph G(V,E), the density of G, $d(G)$, is defined as:

$$D = \frac{2|E|}{|V|(|V|-1)}$$

Definition 2.1.7 (Vertex degree)
The degree k(v) of a vertex v is the number of edges incident on vertex v.

2.2 Directed and undirected graphs

In our field of study we are concerned with different kind of graph. The two main graph families are the directed graph and the undirected ones. The difference between those two classes of graph is the orientation of the edges: in a undirected graph the relation between couples of vetices is symmetric. Otherwise in a directed graph the relation defined by edges is an order couple: the symmetrical propriety of the edges do not exist.

Definition 2.2.1 (Directed Graph)
Let G(V,E) be a graph, $v, w \in V$ elements of G, $\varepsilon(v, w) \in E$ an edge of G. G is directed if the relation defined by $\varepsilon(v, w)$ is an ordered pair and E does not allow duplicate edges.

Definition 2.2.2 (Undirected Graph)
Let G(V,E) be a graph,$v, w \in V$ elements of G,$\varepsilon(v, w) \in E$ an edge of G. G is undirected (or simple) if the relation defined by $\varepsilon(v, w)$ is an unordered pair and E does not allow duplicate edges.

2.3 Weighted graph

A weighted graph is a kind of graph in which every edge or vertex is associated with label and we call this label "weight". Weight is usually a number and it could represent distance, connection or cost. For instance in a weighted graph that represent the topology of a network of routers the weights could means the distance or the traffic congestion between two connected routers.

Definition 2.3.1 (Weighted Graph)
A weighted graph is an ordered tuple (V,E,W,f), where V is the non-empty, finite set of its elements, E is the non-empty, finite set of its edges, W is the finite set of weights such that $|W| > 1$ and f is a discrete function from E to W such that it associates each $e \in E$ to one $w \in W$.

Definition 2.3.2 (Weight of an edge)
Let G(V,E,W,f) be a weighted graph, $\varepsilon_w \in E$ an edge of G labeled with $w \in W$. We call w the weight of ε.

Definition 2.3.3 (Weight of a vertex)
Let G(V,E,W,f) be a weighted graph, $v \in V$ a vertex of G, $\varepsilon \in E(v)$ the edges of v, w_ε the weight of each incident edge of v. The weight of v is defined as the sum of all $w_\varepsilon, \forall \varepsilon \in E(v)$.

Definition 2.3.4 (Weight of a graph)
Let G(V,E,W,f) be a weighted graph, The weight of G is the sum of absolute values of the weight of each edge of G.

$$w = \sum_{e \in E} |w(e)|$$

2.4 Representations

Generally a graph can be represented in two way: adjacency lists or matrix.

Definition 2.4.1 (Adjacency Lists)
Let be G(V,E) a graph. The adjacency lists representation of G associate a list of vertices l for each $v \in V$ such that l contains the neighborhood of v.

Definition 2.4.2 (Adjacency Matrix)
Let be G(V,E) a graph. The adjacency matrix $M \in \mathbb{R}^{N \times N}$ is a representation of G where the generic element

$$m_{i,j} = \begin{cases} 0, & if \nexists \varepsilon(i,j) \wedge i = j \\ 1, & if \exists \varepsilon(i,j) \end{cases} \tag{2.1}$$

.

2.5 Cliques

A clique is an important feature of a graph, it is a subset of V where each vertex is linked with the others. The size of a clique is the number of its vertices: a vertex is a clique of dimension 1, two vertices both connected to the other is a clique of dimention 2 and so on.

Definition 2.5.1 (Clique (complete graph))
Let G(V,E) be an undirected graph, eventually weighted. G is a clique (or complete graph) if D(G) = 1. The following definitions assume that we are intending cliques as complete subgraphs of a graph.

Definition 2.5.2 (Maximal Clique)
Let G(V,E) and $C(V', E')$ be two undirected graphs, with $C \subseteq G$ and C clique. C is a maximal clique if there is no vertex in G that can extend C.

Definition 2.5.3 (Maximum Clique)
Let G(V,E) and $C(V', E')$ be two undirected graphs, with $C \subseteq G$ and C clique. C is a maximum clique if there is no vertex in G that can extend C and for each clique $S_i ofG, |S_i| \leq |C|$.

2.6 Cycles

Another important feature of a graph are the cycles. A cycle is a route inside the graph which begins and ends in the same node.

Definition 2.6.1 (Path)
Let be G(V,E) a graph, a path P in G is a non-empty sequence of vertices of G $\{v_1, v_2, ..., v_k\}, v_i \in V, \forall i \in 1..k$ such that $\exists \varepsilon(v_i, v_{i+1}) \in E$, $1 \leq i \leq k$.

Definition 2.6.2 (Elementary path)
An elementary path in a graph G is a finite path where all the nodes are distinct.

Definition 2.6.3 (Cycle)
Given a graph G(V,E), a cycle C in G is a path $\{v_1, v_2, ..., v_k\}$ such that $\exists \varepsilon(v_k, v_1) \in E$. The length of a cycle is $|C|$.

Definition 2.6.4 (Elementary cycle)
A cycle is elementary if it is a cycle and the sequence of vertices is an elementary path.

Note that in a directed graph a cycle is oriented: in Figure 2.1 we have two similar directed graph, the first has a cycle unlike the second one.

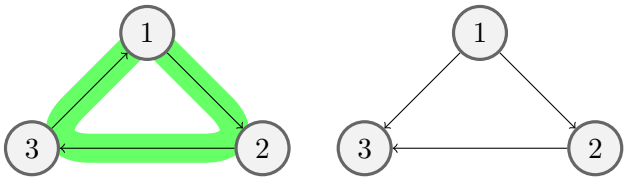

Figure 2.1: Two directed graphs: the first graph has a cycle, the second graph has not any cycle

2.7 Subgraph

Definition 2.7.1 (Subgraph of a graph)
Let $F(V', E')$ and G(V,E) be two graphs. F is subgraph of G, $F \subseteq G$, if and only if $V' \in V$ and $E' \in E$.

Definition 2.7.2 (Weighted subgraph)
Let G(V,E,W,f) and $H(V', E', W', f')$ be two weighted graph. H is subgraph of G, $H \subseteq G$, if and only if:

- $V' \in V$
- $E' \in E$
- $W' \in W$
- *f' is a restriction of f*

The fourth condition could be also written as:

$$\forall \epsilon \in E', f(\epsilon) = f'(\epsilon). \tag{2.2}$$

Definition 2.7.3 (Induced Subgraph)
A graph F(V',E') is a subraph of a graph G(V,E) if $V' \in V$ and $\forall u, v \in V'$ if $\varepsilon(u, v) \in E \Rightarrow \varepsilon(u, v) \in E'$

Note that a subgraph could not be an induced subgraph: in an induced subgraph is required to maintains the original connectivity of each vertices.

Definition 2.7.4 (Connected Component)
Let C(V',E') and G(V,E) be two graphs. C is a connected component of G, $C \subseteq G$ if and only if C is an induced subgraph of G and $\forall v, w \in V', \exists$ a path $P : v, w \in P$.

2.8 Theory Of Complexity

The theory of computational complexity mainly classifies problems according to their inherent difficulty. More precisely this theory classifies problems by the number of necessary operations for solving them with respect to the length of the input. The measures of the difficulty of a problem is given from the number of steps that a *Turing machine*[11] has to make before converging.

Definition 2.8.1 (Computational Complexity)
Let M be a Turing machine on the alphabet A, the (time) complexity of M is the partial function

$$C_M : \mathbb{N} \to \mathbb{N}$$

such that, $\forall n \in \mathbb{N}$, $C_M(n)$ is the maximum number of steps of a convergent computation of M on a input of length $\leq n$.

Let be $\mathcal{C}$ the class of partial functions $f : \mathbb{N} \to \mathbb{N}$ such that:

- $\exists n_0 \in N$: f is defined $\forall n \geq n_0$.
- f is increasing, $\forall n > n_0, f(n) \leq f(n+1)$.

- f is unbounded, $\lim_{n\to\infty} f(n) = +\infty$.

Definition 2.8.2 (Order relation on $\mathcal{C}$)
Given $f, g \in \mathcal{C}$, $f = \mathcal{O}(g) \iff \exists N \in \mathbb{N}, \exists c \in \mathbb{R}, c > 0$, *such that* $\forall n \in \mathbb{N}, n \geq N, f(n) \leq g(n)$.

Definition 2.8.3 (P(PTIME) class)
Let be S a problem, $S \in P$, *where P means polynomial, if there is a Turing machine M and* $w \in S$ *of length n such that:*

- *M converge in w.*
- $C_M = \mathcal{O}(n^k)$ *for some positive integer k.*

In general a problem belongs to the class of P if, in the worst case, it requires a polynomial time with respect to the length of the input to be solved. The P class of problems is a subset of larger set called NP. An intuitive definition of NP class, where N means non-deterministic, is the following. The NP class contains all the problems that can be solved with a hint in a polynomial time.

Definition 2.8.4 (NP class)
Let be S a problem, $S \in NP$ *if there is a non-deterministic Turing machine M and* $w \in S$ *of length n such that:*

- *M converge in w.*
- $C_M = \mathcal{O}(n^k)$ *for some positive integer k.*

The last condition of the *NP* and *P* classes definition do not gives information about the parameter *k*. However the complexity has a fast growth with high values of *k*.

Definition 2.8.5 (NP complete)
Let be S a problem, S is NP-complete if:

- $S \in NP$.
- *S is NP-hard, meaning that* $\forall S' \in NP$ S' *"is faster" than S.*

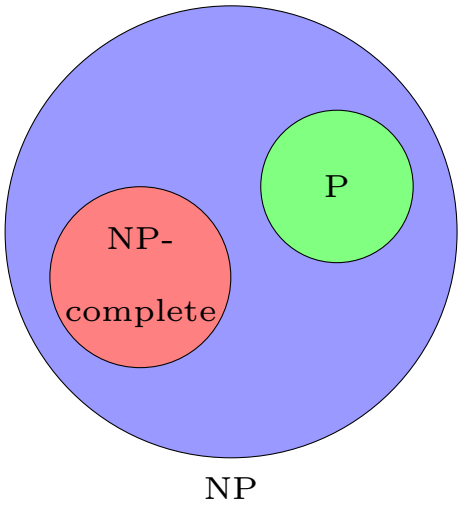

Figure 2.2: Complexity classes

3. Cycles detection: state of art

Now we have the instruments to approach the cycles detection problem:

- **input :** A graph $G(V, E)$.
- **output :** A list of all elementary cycles of G.

In computer science there exists a few problems witch are related with the cycle detection: for instance the deadlock detection for concurrent systems can be solved finding cycles. For each problems of graph theory, the order of the graph $|V|$ is assumed as the length of the input. In our case the output dimension belongs to the numbers of the elementary cycles $|C|$. We know that in a complete graph with n nodes there are

$$\sum_{i=1}^{n-1} \binom{n}{n-i+1}(n-i)!$$

cycles[6]. In this sense the cycle detection problem has a computational complexity, in the worst case, more than polynomial: the problem belong to the class of NP-complete problems.

Several algorithm are able to solve it: we have examined some of the algorithms in our knowledge and we have selected the fastest ones. These algorithms was introduced by Tiernan[10], Tarjan[8], Johnson[6],Szwarcfiter and Lauer[7], Floyd[5] and Brent[3].

3.1 Cycles detection algorithms

The algorithm that we are going to list works with adjacency lists representation of the graph and use a searching methods based on a visit of the graph: every single visit is useful to detect a cycle. The algorithms work only with directed graph. Otherwise all these procedures may found all elementary cycles in an undirected graph through an orientation of the graph; an orientation is a procedure that transform an undirected graph into a directed one preserving the same number of elementary cycles. An undirected graph can be changed into a directed graph by a *DFS* visit: we can give an orientation to each edge crossed in the visit. That method safeguard all the cycle: no cycle is added or deleted from original graph.
In this section we describe how the standard algorithm works and we analyze their computational complexity.

3.1.1 Floyd-Brent algorithm

Floyd's algorithm[5], named "Tortoise and the Hare", is the first algorithm that we consider: it is an iterative procedure allowing to find cycles, in a directed graph, using

two pointer called *tortoise* and *hare*. In his paper Floyd presents an algorithm that works on linked lists: the idea is to let the tortoise and the hare visit the list node by node, following the edges directions. Chosen a start node the pointers can start to go through the same path; naturally the hare is faster then the tortoise, so when they meet in a node it is clear that the hare has done a lap of a cycle. The procedure recives as input the start vertex where it has to initialize the two pointers; for each iteration the *hare* pointer goes two vertices forward following the list. The *tortoise* pointer follows the *hare* node by node: if the pointers meet again the algorithm found a loop, otherwise the iteration goes ahead until it visit the whole list.

Some years later, in 1979, Brent wrote a paper named "An improved Monte Carlo factorization algorithm"[3] suggesting an algorithm based on Floyd's idea. The differences proposed by Brent let the procedure be faster than Floyd's. Brent's algorithm was created to generate pseudorandom numbers, but it is useful also to find a cycles in a graph. The algorithm requires two parameters: $step\_limit > 1$ and $step\_taken$; the last one chosen from a uniform distribution on $[0, 1)$. Those parameters are not crucial for our usage of the algorithm, so in a general case we could set $step\_taken = 0$ and $step\_limit = 2$. The procedure is basically the same as the Floyd one, but at each step of the pointers there is a control of $step\_taken$. If $step\_taken$ reaches $step\_limit$, the tortoise joins the hare and this bound is increased. Brent, in his paper[3], show the improvement of his algorithm, with respect to the Floyd one, giving the following result:

$$W_B \leq 2max\{m, n\} + n \leq W_F, \tag{3.1}$$

where W_B is the complexity of Brent's, W_F the complexity of Floyd's, m and n are the lengths of the path without cycles and the length of the cycle.

Listing 3.1: Brent

```
1  Brent(Node top)
2      turtle := top;
3      rabbit := top;
4      steps_taken := 0;
5      step_limit := 2;
6      forever:
7          if rabbit == end then
8              No Loop Found
9          rabbit := rabbit.next;
10         steps_taken += 1;
11         if rabbit == turtle then
12             return Loop found
13         if steps_taken == step_limit then
14             steps_taken := 0;
15             step_limit *= 2;
16             // teleport the turtle
17             turtle = rabbit
```

Clearly the algorithm looks for a cycle in a linked list: the representation of the graph by adjacency lists let the algorithm work also in a graph. An additional control is required anytime a new cycle is found: we have to check if the current cycle was not already be found. The application of Brent's algorithm has a time complexity in the worst case of $O(|V| + |E|)(|C|)$[3].

3.1.2 Tiernan algorithm

In 1970 James C. Tiernan wrote a paper called *An Efficient Search Algorithm to Find the Elementary Circuits of a Graph*[10]. Tiernan presents an algorithm that works with a *DFS* visit and an array structure to keep the result. During the visit of the graph, two lists P and H are used as data structure: P stores the list of all elementary path, while H stores each cycle closure. The algorithm needs an enumeration of the vertices from 1 to N such that the list $P = \{v_1, ..., v_k\}$ has an increasing order. A vertex v_i can be added to P if $v_i \notin P$ and $v_i > v_1$. Those conditions ensure that the path is elementary and it will be visited once. Moreover H is a $N \times N$ array, entries in the n-th row are closed to vertex n. The entries are made sequentially from left to right in the row and the first zero in a row indicates the end of the closure list. At the beginning all data structures are initialized, i.e. P will contain the first node (named "1") to respect the increasing ordering of vertices. Then the algorithm tries to extend the path, if it is possible, respecting the conditions we mentioned before. Using the adjacency lists, for each enlargement of the path, is easy to check if there is a cycle and report it. Otherwise it checks if, for the root (reported in $P[1]$) all the possible loops have been found, and in this case the procedure is repeated until the whole graph is visited. The computational time to find a single cycle is exponential in the number of nodes[6]: $O(2^{|V|})$ in the worst case.

Listing 3.2: Tiernan

```
1: [Initialize]
    Read N and G.
    P ← 0
    H ← 0
    k ← 1
    P[1] ← 1
2: [Path Extension]
    //for each vertex in V, if possible add a new vertex in the path
    for j ← 1 to N
        search G[P[k],j] such that
        (1) G[P[k],j] > P[1]
        (2) G[P[k],j] ∈ P
        (3) G[P[k],j] ∈ H[P[H],m], m = 1,2,...,N
    if this j is found, extend the path,
        k ← k+1
        P[k] ← G[P[k-1],j]
        go to 2.
    else
        the path cannot be extended.
3: [Circuit Confirmation]
    //check if the current path is a cycle
    if P[1] ∈ G[P[k],j], j = 1,2,...,N than
        no circuits has been formed, go to 4.
    else circuits reported print P.
4: [Vertex Closure]
    if k = 1 then
        all circuits containing vertex P[1] have been considered.
        go to 5.
    else
        H[P[k],m] ← 0, m = 1,2,...,N
    for m | H[P[k-1],m] is leftmost zero in P[k-1] row of H
        H[P[k-1],m] ← P[k]
        P[k] ← 0
        k ← k-1
```

```
35          go to 2.
36  5: [Advanced Initial Vertex]
37      if P[1] = N then
38          go to 6.
39      else
40          P[1] ← P[1] + 1
41          k ← 1
42          H ← 0
43          go to 2.
44  6: [Terminate]
```

3.1.3 Tarjan algorithm

Tarjan[8] introduced in 1973 an evolution of Tiernan[10] procedure. The algorithm enumerates all elementary cycles of a graph using a backtracking method that avoids unnecessary work. It has a bound of $O((|V||E|)(|C|+1))$[10], where $|C|$ is the number of elementary circuits. The algorithm needs a directed graph that meets the following conditions:

- the vertex set has to be enumerated from 1 to N;
- the graph has to be represented with adjacency lists.

Like Tiernan's one, the algorithm creates an elementary path for each vertex s, which is the start node, and that contains no vertex smaller than s. This tecnique allows to discover a cycle as soon as the last vertex is in the path. The algorithm also uses two stacks: one stores the list of paths while the other represent marked nodes via a boolean value (namely *mark*). Tarjan proved that his marking method avoids unnecessary searches that occur in Tiernan's algorithm and this is the main reason of the complexity improvement.

Listing 3.3: Tarjan

```
1  procedure circuit enumeration;
2      begin
3          procedure BACKTRACK (integer v, logical result f);
4              begin
5                  logical g;
6                  f ← false; //check if the path is a cycle
7                  place v on point stack; //the elementary path
8                  mark(v) ← true:
9                  place v on marked stack; //the visited nodes
10                 for w ∈ A(v) do
11                     if w < s then // w can not extend the path
12                         delete w from A(v)
13                     else if w = s then // the path is closed => cycle
14                         begin
15                             output circuit from s to v to s given by
                                   point stack;
16                             f ← true;
17                         end
18                     else if !mark(w) then // w can be added in the path
19                         begin
20                             BACKTRACK (w, g);
21                             f ← f ∨ g;
22                         end;
```

```
23          //f true if an elementary circuit containing the partial path on
                the stack has been found;
24          if f = true then
25              begin
26              a:  while top of marked stack ≠ v do
27                  begin
28                      u ← top of marked stack;
29                      delete u from marked stack;
30                      mark(u)← false;
31                  end;
32                  delete v from marked stack;
33                  mark(v)← false;
34              end;
35          delete v from point stack;
36      end;
37      //Initialize
38      integer n;
39      for i ← 1 until V do
40          mark(i)← false;
41      for s ← 1 until V do
42          begin
43          b:  BACKTRACK (s, flag);
44                  while marked stack not empty do
45                      begin
46                          u ← top of marked stack;
47                          mark(u) ← false;
48                          delete u from marked stack;
49                      end;
50          end;
51      end;
```

3.1.4 Johnson algorithm

In 1975, Tarjan algorithm was improved by Donald B. Johnson with his *circuit finding algorithm*[6] which increases the computational complexity to a bound of $O(|V| + |E|)$ per cycle. Moreover this procedure uses a *DFS* visit and needs an ordering of the vertices and an adjacency list for each vertex. Johnson ensures to avoid some fruitless searches that are included in Tarjan's and Tiernan's algorithm. This improvement belongs to two features in Johnson's algorithm:

- usage of a logic array *"blocked(n)"*: when a node x is added to a elementary path starting in s, it is blocked until every path from x to s intersect the path on an other vertex (different from s);
- a vertex do not become the start node of a path unless it is the last vertex in at least one elementary cycle.

The search of a cycle begins choosing a vertex v as a root, the algorithm go ahead in the subgraph S induced by v where all vertices $n \in S$ are greater than v.

Listing 3.4: Johnson

```
1 begin
2     integer list array A_K(n), B(n);
3     logical array blocked(n);
4     integer s;
```

```
5       logical procedure CIRCUIT (integer v);
6           begin
7               logical f;
8               procedure UNBLOCK (integer u);
9                   begin
10                      blocked(u) := false;
11                      for w ∈ B(u) do
12                          begin
13                              delete w from B(u);
14                              if blocked(w) then UNBLOCK(w);
15                          end
16                  end UNBLOCK
17              f := false;
18              stack v;
19              blocked(v) := true;
20  L1:         for w ∈ A_K(v) do
21                  if w=s then
22                      begin
23                          output circuit composed of stack followed by s;
24                          f := true;
25                      end
26                  else if !blocked(w) then
27                      if CIRCUIT(w) then f := true;
28  L2:         if f then UNBLOCK(v)
29              else
30                  for w ∈ A_K(v) do
31                      if v ∉ B(w) then put v on B(w);
32              unstack v;
33              CIRCUIT := f;
34          end CIRCUIT;
35      empty stack;
36      s := l;
37      while s < n do
38          begin
39              A_K := adjacency structure of strong component K with least
40  vertex in subgraph of G induced by {s, s+ 1, n};
41              if A_K ≠ ∅ then
42                  begin
43                      s := least vertex in V_K;
44                      for i ∈ V_K do
45                          begin
46                              blocked(i) := false;
47                              B(i) := ∅;
48                          end;
49  L3:                 dummy := CIRCUIT(s);
50                      s := s+l;
51                  end
52              else s := n;
53          end
54      end;
```

3.1.5 Szwarcfiter-Lauer algorithm

After the publication of those algorithms, in 1974, a paper by Jayme Szwarcfiter and Peter Lauer was presented. This report named *Finding the elementary cycles of a directed graph in $O(N + M)$ per cycle*[7] show a quantifiable reduction of the computational time from $O(|V||E|)$ to $O(|V| + |E|)$ per cycle. The input of this algorithm is the same of other procedures: a directed graph represented by adjacency lists. The

authors proposes a recursive backtracking procedure that avoids the visit of a path without cycles more than once.

The authors resumes their idea in this example: consider that 1 is the start vertex in the procedure, v is the node at the top of the stack and the edge *(v,w)* is reached.

- If w is unmarked, it is not in the stack: then the elementary path will be extended with w. A new edge from w will be examined.
- If w is marked and not in the stack, w can not be part of a cycle. So w will not be processed again.
- If w is marked and in the stack then a new cycle is found.

Listing 3.5: Szwarcfiter-Lauer

```
1  begin Szwarcfiter-Lauer
2      procedure Cycle(integer v, integer q, integer f)
3      begin
4          integer g, p, w;
5          f := N+1;
6          t := t+1;
7          stack(t) := v;
8          position(v) :=t;
9          mark(v) := true;
10         if reach(v) == N+1 then
11             q := N+1;
12         else if q == N+1 then
13             q := t;
14         p := top(v);
15         while p≠0 do
16         begin
17             w := vertex(p);
18             if reach(w)≥j then
19             begin
20                 if !mark(w) then
21                 begin
22                     Cycle(w,q,g);
23                     if g<f then
24                         f := g;
25                 end
26                 else if position(w)<q then
27                 begin
28                     output cycle from v to w from stack;
29                     f := position(w);
30                 end
31             end
32             p := link(p);
33         end
34             if position(v)≥f then
35             mark(v) := false;
36         t := t-1;
37         reach(v) := j;
38         position(v) := N+1;
39     end Cycle;
40     // Initialize
41     integer array reach, top, stack, position(1::N);
42     logical array mark(1::N);
43     integer array vertex, link(!::M);
44     integer j,t, dummy;
```

```
45      t := 0;
46      for j := 1 until N do
47      begin
48          top(j) := 0;
49          mark(j) := false;
50          position(j) := reach(j) := N+1;
51      end
52      read the graph and construct the adjacency lists;
53      for j := 1 until N do
54          if reach(j) == N+1 then
55              Cycle(j, dummy, dummy);
56  end Szwarcfiter-Lauer
```

3.1.6 Computational complexity

The table reports the computational complexity, with descending order, of each algorithm.

Tiernan	$O(2^{\|V\|})$
Tarjan	$O(\|V\|\|E\|)(\|C\|+1))$
Floyd-Brent	$O(\|V\|+\|E\|)(\|C\|))$
Johnson	$O(\|V\|+\|E\|)(\|C\|))$
Szwarcfiter-Lauer	$O(\|V\|+\|E\|)(\|C\|))$

Table 3.1: Computational complexity

4. A brief introduction to algebraic and computational topology

4.1 Computational topology

Topology is a branch of geometry for studying shapes: the basic idea is that it allows a set of possible transformations, more precisely a group of homeomorphism, that preserv the topology of an object. This group of possible transformation is less restrictive than *Euclidean* one. For instance, topology admits stretching and shrinking; on the contrary, cutting and gluing transformations are forbidden. A well-known example of homeomorphism is that a coffee mug and a donut are topologically the same. In fact the mug and the donut have the same topological invariant: a hole that is preserved during the transformation.

Figure 4.1: Cup to Torus

Definition 4.1.1 (Topology)
A topology on a set X is a subset $T \subseteq 2^X$ s.t. :

- *If $S_1, S_2 \in T$, then $S_1 \cap S_2 \in T$.*
- *If $\{S_J : j \in J\} \subseteq T$, then $\cup_{j \in J} S_j \in T$.*
- $\emptyset, X \in T$.

A topology **T** of a set **X** is a system of sets that describes the connectivity of **X**. Those can be either *open* or *closed*. If $S \in T$, **S** is an *open* set. The *closed* sets are $X - S$. A set may be closed, open, both or neither. Combining a set with topology we get the spaces we are interested in:

Definition 4.1.2 (Topological space)
A topological space $\mathbb{X}$ is the pair (X, T) where T is the topology of the set X.

Definition 4.1.3 (Interior, closure and boundary)
The interior $\dot{A}$ of a set $A \subseteq X$ is the union of all open sets conteined in A. The closure $\bar{A}$ of a set $A \subseteq X$ is the intersection of all closed sets containing A. The boundary of a set A is $\partial A = \bar{A} - \dot{A}$.

Definition 4.1.4 (Neighborhoods)
A neighborhoods of $x \in X$ is any $A \subseteq X$ such that $x \in \dot{A}$.

Definition 4.1.5 (Homeomorphism)
A homeomorphism $f : \mathbb{X} \to \mathbb{Y}$ is a bijection such that both f and f^{-1} are continuous. We say that $\mathbb{X}$ is homeomorphic to $\mathbb{Y}, \mathbb{X} \approx \mathbb{Y}$, and that $\mathbb{X}and\mathbb{Y}$ have the same topological type.

4.2 Simplicial complex

Simplicial complexes are a tool to represent a topological space. A simplicial complex K is a system of sets closed under the subset relation such that if $\sigma \in K$ and $\tau \subseteq \sigma$, then $\tau \in K$. Let $u_0, u_1, ..., u_k$ be points of $\mathbb{R}^d$. A point $x = \sum_{i=0}^{k} \lambda_i u_i$ is an affine combination of the u_i points if $\sum_{i=0}^{k} \lambda_i = 1$. The affine hull is the set of affine combinations. It is a k-plane if the k+1 points are affinely independent, by which we mean that any two affine combinations, $x = \sum \lambda_i u_i$ and $y = \sum \mu_i u_i$, are the same if and only if $\lambda_i = \mu_i$ for all i. The k+1 points are affinely independent if and only if the k vectors $u_i - u_0$, for $1 \leq i \leq k$, are linearly independent. In $\mathbb{R}^d$ we can have at most d linearly independent vectors and therefore at most *d+1* affinely independent points. An affine combination $x = \sum \lambda_i u_i$ is a *convex combination* if all λ_i are non-negative. The *convex hull* is the set of convex combinations.

Definition 4.2.1 (k-simplex)
A k-simplex is the convex hull of k+1 affinely indipendents points, $\sigma = conv\{u_o, u_1, ..., u_k\}$. Its dimension is $dim\sigma = k$.

For instance, a *vertex* is a 0-simplex, an *edge* a 1-simplex, a *triangle* a 2-simplex and so on.

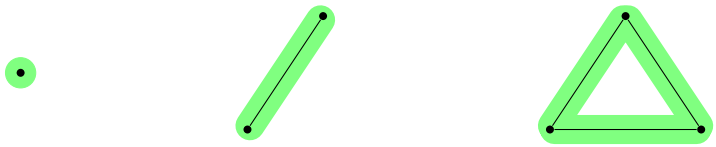

Figure 4.2: A 0-simplex, a 1-simplex and a 2-simplex

Definition 4.2.2 (Face)
A face of σ is the convex hull of a non-empty subset of the u_i and it is proper if the subset is not the entire set.

Definition 4.2.3 (Simplicial complex)
A simplicial complex is a finite set of simplices K such that $\sigma \in K$ and $\tau \leq \sigma$ implies $\tau \in K$, and $\sigma, \sigma_0 \in K$ implies $\sigma \cap \sigma_0$ is either empty or a face or both.

Definition 4.2.4 (Subcomplex)
A subcomplex of a simplicial comples K is a simplicial complex $L \subseteq K$

Definition 4.2.5 (n-skeleton)
A n-skeleton of a simplicial complex K is a subcomplex $L \subseteq K$ such that $L = \{\sigma \in K | dim(\sigma) \leq n\}$

The 1-skeleton of a simplicial complex is a graph.

Definition 4.2.6 (Filtration)
A filtration of a complex K is a nested sequence of subcomplex, $\emptyset = K^0 \subseteq K^1 \subseteq K^2 \subseteq .... \subseteq K^m = K$. We call a complex K with a filtration a filtered complex.

There exists a few representation of a simplicial complex. We are interested in a particular complex that is based on a graph: the *Vietoris-Rips* one.

Definition 4.2.7 (Vietoris-Rips Complex)
Given $S \subseteq \mathbb{Y}$ and $\varepsilon \in \mathbb{R}$, let $G_\varepsilon = (S, E_\varepsilon)$ be the ε-neighborhood graph of S, where

$$E_\varepsilon = \{\{u, v\} | d(u, v) \leq \varepsilon, u \neq v \in S\}$$

, the Vietoris-Rips complex V_ε is the clique complex of the ε-neighborhood graph.

For instance, given a undirected graph we can create a clique complex by the ε-neighborhood graph, wehre each maximal clique becomes a maximal simplex.

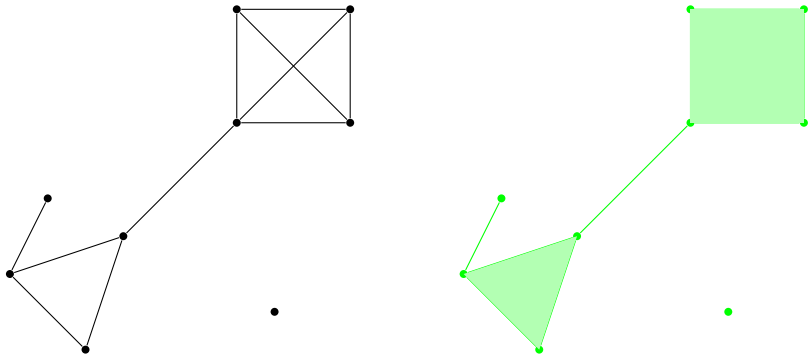

Figure 4.3: In green a clique complex based on the black graph.

4.3 Homology

Given any topological space,a *topological invariant* is a function that assign the same object to homeomorphic spaces. The *homology* is a topological invariant that assign a group to a topological space. The best quality of homology is that it is feasible: there exists fast algorithms that associate a finite number of Abelian groups to a topological space. Now we introduce the simplicial homology that is a lower form of homology. From the one part we have an homology computable in a feasible time, at the other part we pay for this velocity: homology is a less power invariant than homotopy. However we have a kind of homology that works with simplicial complexes. To define homology groups, we need simplicial analogs of paths and loops.

Definition 4.3.1 (Group)
*A group $< G, * >$ is a set G with a binary operation * such that:*

- $\forall a, b, c \in G, a * (b * c) = (a * b) * c$.
- $\forall a \in G, \exists e : a * e = e * a = a$.
- $\forall a \in G, \exists a' : a * a' = a' * a = e$.

Definition 4.3.2 (Chain group)
The kth chain group of a simplicial complex K is $\langle C_k(K), + \rangle$, the free Abelian group on the oriented k-simplices, where $[\sigma] = -[\tau]$ if $\sigma = \tau$ and σ and τ have different orientations. An element of $C_k(K)$ is a k-chain, $\sum_q n_q[\sigma_q], n_q \in \mathbb{Z}, \sigma_q \in K$.

Definition 4.3.3 (Boundary homomorphism)
Let K be a simplicial complex and $\sigma \in K, \sigma = [v_0, v_1, ..., v_k]$ The boundary homomorphism $\partial_k : C_k(K) \to C_{k-1}(K)$ is

$$\partial_k \sigma = \sum_i (-1)^i [v_0, v_1, ..., \widehat{v_i}, ..., v_n]$$

where $\widehat{v_i}$ indicates that v_i is deleted from the sequence.

Definition 4.3.4 (Cycle and boundary)
The k-th cycle group is $Z_k = ker\partial_k$. A chain that is an element of Z_k is a k-cycle. The k-th boundary group is $B_k = im\partial_{k+1}$. A chain that is an element of B_k is a k-boundary. We also call boundaries bounding cycles and cycles not in B_k non-bounding cycles.

4.3.1 Simplicial homology

Definition 4.3.5 (Homology group)
The k-th homology group is

$$H_k = Z_k / B_k = ker\partial_k / im\partial_{k+1}$$

If $z_1 = z_2 + B_k, z_1, z_2 \in Z_k$, we say z_1 and z_2 are homologous and denote it with $z_1 \sim z_2$ The *k-th homology group* of a topological space $\mathbb{X}$ measures the numbers of k-dimension holes in $\mathbb{X}$.

Note that our problem is to find all the elementary cycles in a graph. It is easy to see a similarity between a cycle and a hole of 1-dimension. Therefore we focus on H_1 homology group that includes all 1-dimension holes. However the group H_K depends on its lower dimension group so it is necessary to calculate also the H_0 group. This weak correspondence among cycles and holes is our starting point to face up the cycle detection problem with the homology. A further question that we have to answer is: are the 1-dimension holes all the elementary cycles in a graph?

Betti numbers describe the topology of a growing simplicial complex by a sequence of integers. Our hope is that these numbers contain topological information about the original space. Unfortunately our representation scheme generates a lot of additional topological attributes, namely topological noise, all of which are captured by homology. We cannot distinguish between the features of the original space and the noise spawned by representation.

Definition 4.3.6 (k-th Betti number)
The k-th Betti number $\mathrm{B_k}$ *of a simplicial complex K is* $\mathrm{B(H_k)}$ *a way to count the number of hole in different dimensions.*

4.3.2 Persistent homology

The solution that makes us able to catch only significant features rather than the topological noise is the persistent homology. Persistent homology lists the homology classes during the filtration: at what value of filtration does a hole appear and how long does it persist until it is filled? We consider all the holes that appear among the filtration as noise. Once the the filtration finished the resulted holes are called persistent.

Definition 4.3.7 (Homology of filtration)
Let K^l be a filtration of a space $\mathbb{X}$. Let $Z_k^l = Z_k(K^l)$ and $B_k^l = B_k(K^l)$ be the kth cycle

and boundary group of K^l, respectively. The k-th homology group of K^l is $H_l^l = Z_k^l/B_k^l$. The k-th Betti number β_k^l of K^l is the rank of H_k^l.

Definition 4.3.8 (Persistence)
Let z be a nonbounding k-cycle that is created at time i by simplex σ, and let $z' \sim z$ be a homologous k-cycle that is turned into a boundary at time j by simplex τ. The persistence of z, and its homology class $[z]$, is j-i-1. σ is the creator and τ is the destroyer of $[z]$. We say that τ destroys z and the cycle class $[z]$. We also call a creator a positive simplex and a destroyer a negative simplex. If a cycle class does not have a destroyer, its persistece is ∞.

Often, a filtration has an associated map $\rho : S(K) \to \mathbb{R}$, which maps simplices in the final complex to real numbers. We may also define persistence in terms of the birth times of two simplices: $\rho(\sigma_j) - \rho(\sigma_i)$.

Definition 4.3.9 (Time-based persistence)
Let K be a simplicial complex and let $K^\rho = \{\sigma^i \in K | \rho(\sigma^i) \leq \rho\}$ be a filtration defined for an associated function $\rho : S(K) \to \mathbb{R}$. Then for every real $\pi \geq 0$, the π-persistent k-th homology group of K^ρ is

$$H_k^{\rho,\pi} = Z_k^\rho / (B_k^{\rho+\pi} \cap Z_k^\rho) \tag{4.1}$$

The π-persistent k-th Betti number $\beta_k^{\rho,\pi}$ of K^ρ is the rank of $H_k^{\rho,\pi}$. The persistence of a k-cycle, created at time ρ_i and destroyed at time ρ_j, is $\rho_j - \rho_i$.

4.4 From graph to simplicial complexes

The brief introduction about topology and persistent homology gives us the concepts and the reasons to manipulate a graph as a topological object. Moreover we have to clarify how persistent homology can help us to detect a cycle. Intuitively we can calculate the 1-homology group from a simplicial complex: all the found objects are 1-dimensional holes bounded by simplices.

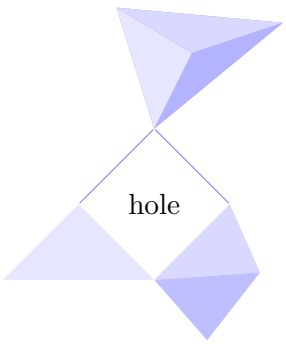

Figure 4.4: A clique complex with a 1-dimension hole

For instance, the clique complex in the Figure 4.4 draws attention to the unique hole bounded by four cliques. With this approach there is not a full correspondence between holes and cycles. The reason of this fault derives from the construction of the simplicial complex. For example in a clique complex a clique of three vertices (a triangle) is a full face whereas in graph theory it is a cycle. So we have to find a better to convert a graph to a simplicial complex, so that we can preserve the cycle structure without hiding any of them. We reduce the clique complex by considering as simplices

only edges and vertices. Triangles, tetrahedra and higher dimension cliques will not be more seen as simplices, but as a combination of edges. The problem about full faces is now avoided and we get a complete mapping among holes and cycles. We shift from a graph to a simplicial complex as follows. Given an undirected graph $G(V, E)$ our complex $K(G)$ of an undirected graph G is formed by the sets of vertices in the edges of G. Note that the 1-skeleton of $K(G)$ is really G.

4.4.1 Simplicial complex filtration

The computation of persistent homology requires a filtration of the complex. We filter the graph by the weight of its edges. Given an ordered list of weights $w_1, w_2, ..., w_k$ we present at time zero all the 0-simplices (vertices) and at time i all the 1-simplices (the edges) of weight w_i. In case of a unweighted graph we assume an unitary weight for each edge.

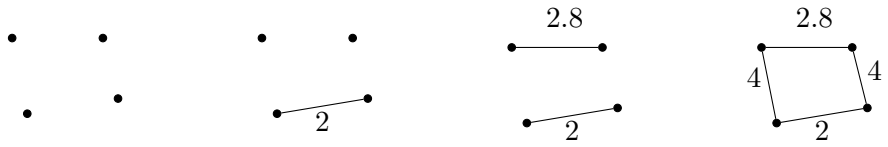

Figure 4.5: Filtration of a simplicial complex

4.5 Computing persistent homology

In this section, we look at one of the algorithms for computing persistent homology. We put our focus on an algorithm for computing Betti numbers by Delfinado and Edelsbrunner (1995). This algorithm works over subspaces of $\mathbb{S}^3$, which do not have torsion. This algorithm works simplices as positive or negative. We also show how the algorithm may be used to speed up the computation of persistence.

4.5.1 A first algorithm for computing Betti numbers

We assume that the input spaces are three-dimensional and torsion-free. Consequently, we use $\mathbb{Z}^2$ coefficients for computation: these coefficients greatly simplifies homology. The homology groups are vector spaces, a k-chain is simply the list of simplices with coefficients 1, each simplex is its own inverse. The only nonzero Betti numbers to be computed are β_0, β_1, and β_2. We use a total ordering to construct filtration: only one simplex is added at each time step, namely, $K^i = \sigma^j | 0 \leq j \leq i$, for $0 \leq i < m$. Simplex σ^i is added at time i, so its index is also its birth index. The total ordering of simplices in a filtration permits a simple incremental algorithm for computing Betti numbers of all complexes in a filtration. Before running the algorithm, the Betti number variables are set to the Betti numbers of the empty complex, (all variables equal to 0). The algorithm returns a list of 3 integers $INTEGER^3$. The main part of the algorithm is the task in which we decide if a (k+1)-simplex belongs or does not to a (k+1)-cycle. For $k+1 = 0$, this is trivial because every vertex belongs to a 0-cycle. For edges, we maintain the connected components of the complex, each represented by its vertex set. An edge belongs to a 1-cycle iff its two endpoints belong to the same component. Triangles and tetrahedron are treated similarly, using the symmetry provided by complementarity,

duality, and time-reversal. We use these algorithms to mark the simplices as positive or negative. Let $pos_k = pos_k^l$ and $neg_k = neg_k^l$ be the number of positive and negative k-simplices in K^l . The correctness of the incremental algorithm implies:

$$\beta_k = pos_k - neg_{k+1} \tag{4.2}$$

for $0 \leq k \leq 2$. Or in natural language, the Betti number β_k is the number of k-simplices that create k-cycles minus the number of (k+1)-simplices that destroy k-cycles. This algorithm is the basis for understanding how the persistent homology is computed over more complicated structures like $\mathbb{R}^d$, *PID*, *fields*.

Listing 4.1: Betti-Numbers

```
1  integer^3 BETTI-NUMBERS(){
2      for i = 0 to m - 1{
3          k=dim σ^i -1;
4          if σ^i belongs to a (k+1)-cycle in K^i
5            β_{k+1} = β_{k+1} + 1;
6          else
7            β_k = β_k - 1;
8      }
9      return(β_0, β_1, β_2);
10 }
```

The computational complexity in the worst case is $O(n^3)$. The computational complexity is wasted by the *checking* step in which the subroutine checks if a simplex belongs to a cycle or not.

The previous algorithm has been implemented in the *Java* library *javaplex*[9].

5. Cycle detection algorithm: a persistent homology application

An alternative solution to the cycle detection problem is presented in this chapter. We show how to use the persistent homology to find all elementary cycles in a graph. We exhibit two different kind of algorithm: first of all we present the algorithm for undirected graphs, after that we focus on directed graphs that requires more details. In our algorithms we assume the input to be a graph without self-loops (edges that connect node to itself) and we do not consider cycles of dimension two. A cycle with two edges is possible only in a directed graph and it consist in two nodes connected each other by two edges. Furthermore, let be $G(V, E)$ a graph and s a simplex :

- $w : E \to \mathbb{R}^+$, such that $\forall e \in E, w(e)$ is the weight of the edge e.
- $weightIndex : \mathbb{R} \to \mathbb{N}$, such that $weightIndex(x)$ is the filtration index of the weight of an edge.
- if $dim(s) = 1$ s is equal to a vertex.
- if $dim(s) = 2$ s is an edge where $s[0]$ and $s[1]$ are the vertices connected by the edge.

5.1 Undirected graphs algorithm

In this paragraph we present two implementations of the algorithm for undirected graph: the first implementation is for unweighted graphs, the second implementation is for undirected weighted graphs. The main difference is regarding how do we build the simplicial complexes.

5.1.1 Unweighted

Listing 5.1: UndirectedCyclesFinder

```
1  UndirectedCyclesFinder(UndirectedGraph G(V,E)){
2      SimplicialComplex complex;
3      List<Simplex> generators;
4      //Add to the complex all the 0-simplices
5      for each v ∈ V
6          complex.add(v, 0);
7      //Add to the complex all the 1-simplices
8      integer i := 0;
9      for each e ∈ E
10         complex.add(e, i);
```

```
11          i := i+1;
12      //Compute the persistent homology keeping just persistent 1-dimension
            holes
13      generators :=
            computePersistentHomology(complex).getPersistentGeneratorsAtDim(1);
14      return generators;
15  }
```

5.1.2 Weighted

Listing 5.2: UndirectedCyclesFinder

```
1  UndirectedCyclesFinder(UndirectedWeightedGraph G(V,E,W,w)){
2      SimplicialComplex complex;
3      List<Simplex> generators;
4      //Add to the complex all the 0-simplices
5      for each v ∈ V
6          complex.add(v, 0);
7      //Add to the complex all the 1-simplices
8      for each e ∈ E
9          complex.add(e, weightIndex(w(e)));
10     //Compute the persistent homology keeping just persistent 1-dimension
           holes
11     generators :=
           computePersistentHomology(complex).getPersistentGeneratorsAtDim(1);
12     return generators;
13 }
```

For those algorithms the computational complexity is immediate: the two for-cycles loop at least $|E|$ times making a $O(1)$ operation. The heaviest computation regards the calculation of persistent homology that, in the worst case, has a time complexity of $O(|V|^3)$.

5.2 Directed graphs algorithm

As we saw in previous chapters there is a full correspondence between an undirected graph and a simplicial complex, because vertices and edges are both cliques. This feature is missing in a directed graph where an oriented edge is not a clique. In other words, if we present a directed edge as a simplex we loose the information about its orientation because the simplex $\{a, b\}$ is equal to the $\{b, a\}$ one. For instance a simplicial complex of the graph in the Figure 5.1 underline an holes (a cycle) that not exist in the graph. Our approach to this problem is to leave the construction of the simplicial

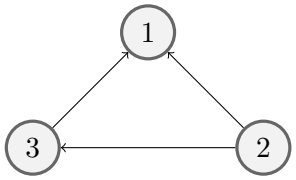

Figure 5.1: Directed graph without cycles

complex as before. To avoid the wrong cycle that we may found, we make a test for each cycles and discard the fake ones.

5.2.1 Unweighted

Listing 5.3: DirectedCyclesFinder

```
1  DrectedCyclesFinder(DirectedGraph G(V,E)){
2      SimplicialComplex complex;
3      List<Simplex> generators;
4      for each v ∈ V
5          complex.add(v , 0);
6      integer i := 0;
7      for each e ∈ E
8          complex.add(e, i);
9          i := i+1;
10     generators :=
           computePersistentHomology(complex).getPersistentGeneratorsAtDim(1);
11     //Remove all non cycles
12     return checkCycles(generators);
13 }
```

5.2.2 Weighted

Listing 5.4: DirectedCyclesFinder

```
1  DrectedCyclesFinder(DirectedGraph G(V,E)){
2      SimplicialComplex complex;
3      List<Simplex> generators;
4      for each v ∈ V
5          complex.add(v , 0);
6      integer i := 0;
7      for each e ∈ E
8          complex.add(e, i);
9          i := i+1;
10     //Remove all non cycles
11     generators :=
           computePersistentHomology(complex).getPersistentGeneratorsAtDim(1);
12     return checkCycles(generators);
13 }
```

5.2.3 Cycles control

The method to check if a set of edges is really a cycle in the graph is the following. It is based on the idea that a cycle has at least one edge with the same start node. So for each simplex in a generators we orient it as the edge that it describe. If the current start vertex was already present the cycle it is deleted.

Listing 5.5: checkCyckes

```
1  checkCycles(List<Simplex> generators){
2      for each List<Simplex> cycle ∈ generators
3          for each Simplex s ∈ cycle
4              Set<V> vertices; //The visited vertices
5              // Check if s has the same orientation of the edge it describe
6              if ∃ e ∈ E : s[0]==e.source & s[1]==e.target then
7                  if vertices.contains(s[0]) then
8                      //delete the fake cycle
9                      generators.remove(s);
10                     continue;
```

```
11                  vertices.add(s[0]);
12              else
13                  if vertices.contains(s[1]) then
14                      //delete the fake cycle
15                      generators.remove(s);
16                      continue;
17                  vertices.add(s[1]);
18      return generators;
19  }
```

The first part of the algorithm Listing 5.3 has the same complexity than before: $O(|V|^3)$. Instead, the *checkCycle* method require a longer time. The external for-cycle check $|C|$ times if the current generator is a cycle. Control the correctness of the cycle has a complexity depending on the length of each cycle $O(|E|)$. In conclusion the total computational complexity of the *Directed Cycle Finder* is $O(|C||E|)$.

5.2.4 Computational complexity

DirectedCycleFinder	$O(\|C\|\|E\|)$
UndirectedCycleFinder	$O(\|V\|^3)$

Table 5.1: Computational complexity

The theoretical results given from the computational complexity analysis show a clear improvement of the complexity. In particular the *UndirectedCyclesFinder* algorithm is a really feasible method: its complexity is polynomial with respect to the number of vertices and is independent from the number of the cycles. The next step is a comparison among the implementations of our algorithms and the classic ones.

5.3 Implementation

We implemented our algorithms and the classic ones, in *Java*. The source codes of *Tiernan, Tarjan, Johnson* and *Szwarcfiter-Laurel* are taken from the *Java* library *jGraphT*[1]. We wrote the code for *Floyd-Brent* and our algorithms. The development of our algorithms was integrated in the *JHoles*[2]: a tool for understanding biological complex networks. Our persistent homology engine is *javaPlex*, a *Java* library that offers all the needed methods to compute persistent. We also chose *JGraphT* as data structure to handle graphs.

5.3.1 From undirected graph to directed graph

In order to compare our *UndirectedCyclesFinder* with the classic algorithms we decided to orient the undirected graph using a *DFS* visit. We chose this method because it keep the original cycles and create a directed graph with the same order and size. So we can assume that the resulting graph entails the same computational complexity to the cycles detection. The algorith works in this way: for each vertex of the directed graph visited by the *DFS* it is added in the directed graph. Concurrently the directed graph obtain a new edge that start from the last visited vertex and end to the current one. The directed graph obtained by the procedure keep the same cycles of the undirected graph. In fact the *DFS* sketch a tree of the graph and it never create an edge from the

same start vertex twice. The running time for the orientation of the graph is added to each classic algorithm, it has a minimal effect to the total results: the deep first visit has a computational complexity of $O(|V| + |E|)$ in the worst case.

Listing 5.6: iterativeDFS

```
1 itertiveDFS(UndirectedGraph G(V,E), V root){
2     DirectedGraph D(V,E);
3     //Choose v as random root vertex.
4     V v := root;
5     V w ;
6     //stack contains all visited vertices with some neighbors not yet
          considered.
7     Stack stack := ∅;
8     Set notVisited := V\{v};
9     stack.push(v);
10    while (stack != ∅) {
11        //take start vertex
12        v := stack.peek();
13        graph.addVertex(v);
14        if (∃w ∈ A(v): w was never considered) {
15            //w is the destination vertex.
16            w = it.next();
17            if (w ∈ notVisited) {
18                notVisited.remove(w);
19                stack.push(w);
20            }
21            //insert new edge in graph
22            graph.addVertex(w);
23            graph.addEdge(v, w);
24        }
25        //v has no more neighbors to consider or hasn't neighbors.
26        else{
27            stack.pop();
28        }
29    }
30    return D;
31 }
```

6. Benchmarks

In this chapter we present the results given from the executions of the algorithms on a sets of graphs. The laptop that has been used for executing our tests is:

Platform

- ASUS K56CM notebook with a dual-core processor
- CPU Intel Core i5 3317U @1.7GHz
- Ram 2X2GB DDR3 @1600MHz SDRAM
- OS Windows 8.1 pro 64bit

6.1 Dataset

We generated a set of random graphs, namely á la Erdős–Rényi [4], with incremental size and order. The networks are all undirected graph represented by an *edgelist*: a text file where in each row is formatted as follow:

$$Vertex_i \quad Vertex_j \quad Weight_{i,j}$$

The graphs range go from 8 vertices, 5 edges and 0 cycles to 5000 nodes, 20000 arcs and 15000 elementary cycles Figure 6.1.

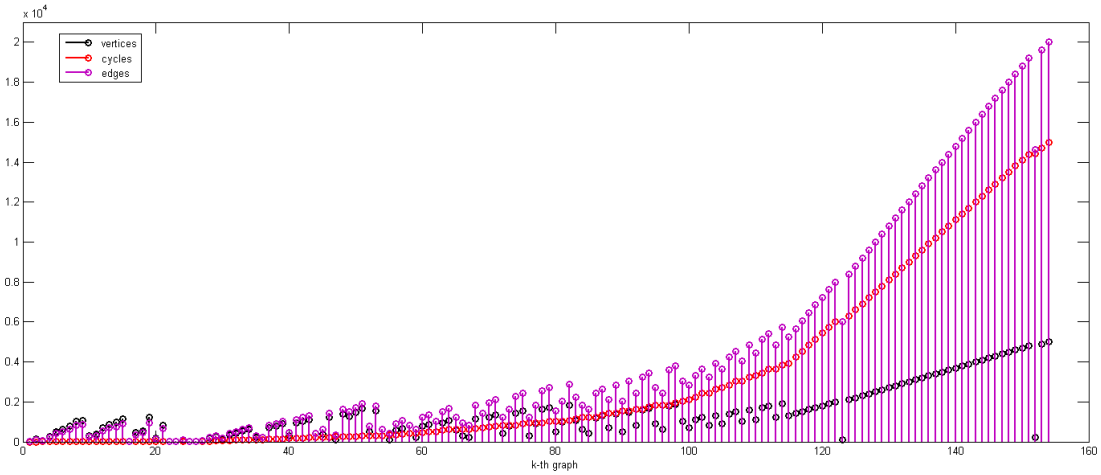

Figure 6.1: Random Graphs Measures: the red stems are the number of cycles per graph, the purple stems are the number of edges per graph, the black stems are the number of nodes per graph.

6.2 Results

The following charts show the benchmark results on the dataset and they are the trend of the execution time (in seconds) depending on the number of cycles. The time axis has a logarithmic scale in order to reduce the gap among the functions and keep all of them in the same picture. A policy has been introduced, it fixed up to 30 minutes the upper limit for the time execution. The *Johnson* algorithm is the only one that was stopped after it reached the time limit.

6.2.1 Directed graphs results

The semi log-plots, Figure 6.2, show the execution time occurred for detect all the elementary cycles in the graphs of the dataset which were oriented by the *DFS*. From the analysis of the chart we can assert that the order given by the theoretical complexity, Table 3.1, is lost, i.e. *Johnson* algorithm has the worst performance. The trends of the curves decrease several times: the most evident cusp occurs approximately at 6000 cycles. The reason of those changes come from the computed graph: the graph coresponding to the lower value of the chart has a low number of vertices and an high density in respect with the other graphs ($d = 0.5$). The low number of nodes is the reason of those lover bounds in the chart.
In regard to our algorithm the *DirectedCyclesFinder* is slower than *Szwarcfiter-Laurel*; this is an expected result because our algorithm spent many time to check each cycle. However a feature of the *DirectedCyclesFinder* is that it keep a constant growth with respect to the number of cycles.

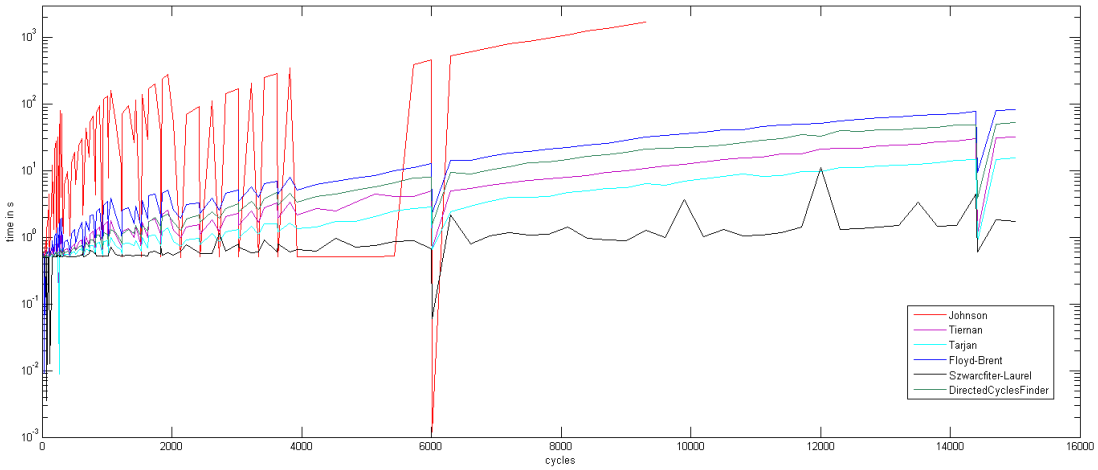

Figure 6.2: Directed Graphs Comparison

6.2.2 Undirected graphs results

The chart in Figure 6.3 concern the results of the executions times with as input the undirected graphs. The execution times includes the time needed for the orientation of the graph, except for the *UndirectedCyclesFinder*. Note that the orientation do not require more than 90ms for the biggest graph. The main difference with the previous chart regards our algorithm: the *UndirectedCyclesFinder* result a very efficient method such as *Szwarcfiter-Laurel*.

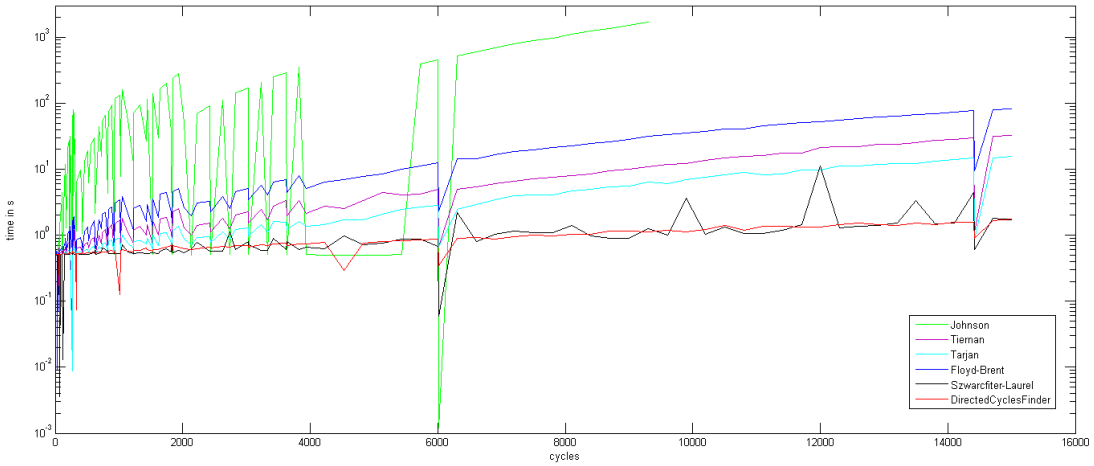

Figure 6.3: Undirected Graphs Comparison

In the last chart, Figure 6.4, we reported the *UndirectedCyclesFinder* curve and the *Szwarcfiter-Laurel* one in order to analyze only those two algorithms. As for the *DirectedCyclesFinder* the trend of the *UndirectedCyclesFinder* is constant. Instead the *Szwarcfiter-Laurel* algorithm shows in some points of the chart a deterioration of the performance. In this case those points are not related to a particular configuration of the graph. However our algorithm keep a trend more safe without fluctuations.

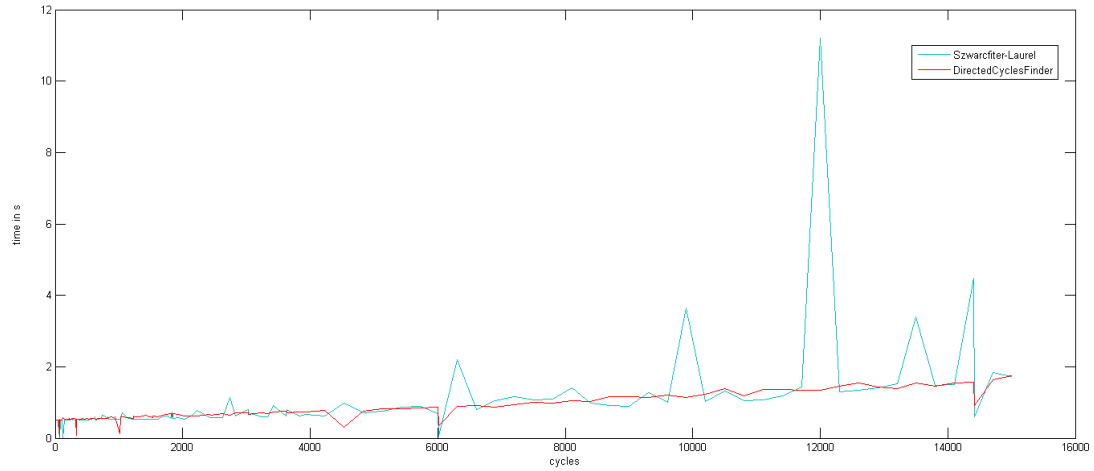

Figure 6.4: Best Performance Algorithms Comparison

Conclusions and further development

We faced-up the problem regarding the detection of all elementary cycles in a graph. We started analysing the domain of the problem and choosing a set of existing algorithms: the fastest in our knowledge. Those algorithms have a classical approach and belongs to the class of *NP-complete* problems.
Then we introduced the employment of computational topology to manipulate graphs as topological objects. We showed our method to transform a graph to a simplicial complex in order to calculate on it the persistent homology. We used persistent homology to find all elementary cycles in a graph in a feasible way: the worst case complexity is $O(n^3)$ for an undirected graph. Then we developed two algorithms: one for directed graph (*DirectedCycleFinder*) an one for undirected one (*UndirectedCycleFinder*) which have respectively a computational complexity of $O(|E||C|)$ and $O(|V|^3)$. To validate our assumptions we tested the implementation in *Java* of classical algorithms and our ones over a set of graphs with crescent dimensions. The results of the test show that on an undirected graph the *UndirectedCycleFinder* works faster than the others and its execution time increase linearly with respect to the size of the graphs. The *DirectedCycleFinder* has not the best execution time instead it has to check every single cycle founded by the homology, however it is a feasible algorithm.
As further goal we want to analyze those algorithm in a more detailed way using a code profiler. This analysis give us statistics about the usage of *CPU*, *RAM* and executions times also for each method of the algorithms. Moreover we want to continue the study for a more efficient algorithm that works on the directed graphs in order to have a method efficient like the one which works on undirected graphs.

Bibliography

[1] Jgrapht, free java graph library. URL http://jgrapht.org/.

[2] Jacopo Binchi, Emanuela Merelli, Matteo Rucco, Giovanni Petri, and Francesco Vaccarino. Jholes: A tool for understanding biological complex networks via clique weight rank persistent homology.

[3] RichardP. Brent. An improved monte carlo factorization algorithm. *BIT Numerical Mathematics*, 20(2):176–184, 1980. ISSN 0006-3835. doi: 10.1007/BF01933190. URL http://dx.doi.org/10.1007/BF01933190.

[4] László Erdős, Antti Knowles, Horng-Tzer Yau, and Jun Yin. Spectral statistics of erdős-rényi graphs ii: Eigenvalue spacing and the extreme eigenvalues. *Communications in Mathematical Physics*, 314(3):587–640, 2012.

[5] Robert W. Floyd. Nondeterministic algorithms. *J. ACM*, 14(4):636–644, October 1967. ISSN 0004-5411. doi: 10.1145/321420.321422. URL http://doi.acm.org/10.1145/321420.321422.

[6] D. Johnson. Finding all the elementary circuits of a directed graph. *SIAM Journal on Computing*, 4(1):77–84, 1975. doi: 10.1137/0204007. URL http://dx.doi.org/10.1137/0204007.

[7] Jayme L. Szwarcfiter and Peter E. Lauer. Finding the elementary cycles of a graph in o(n+m) per cycle. *Technical Report Series*, (60), 1974. URL www.cs.ncl.ac.uk/publications/trs/papers/60.pdf.

[8] R. Tarjan. Enumeration of the elementary circuits of a directed graph. *SIAM Journal on Computing*, 2(3):211–216, 1973. doi: 10.1137/0202017. URL http://dx.doi.org/10.1137/0202017.

[9] Andrew Tausz, Mikael Vejdemo-Johansson, and Henry Adams. Javaplex: A research software package for persistent (co)homology, 2011. URL http://javaplex.github.io/.

[10] James C. Tiernan. An efficient search algorithm to find the elementary circuits of a graph. *Commun. ACM*, 13(12):722–726, December 1970. ISSN 0001-0782. doi: 10.1145/362814.362819. URL http://doi.acm.org/10.1145/362814.362819.

[11] Alan Mathison Turing. On computable numbers, with an application to the entscheidungsproblem. *J. of Math*, 58:345–363, 1936.

[12] Wikipedia. Travelling salesman problem — wikipedia, the free encyclopedia, 2014. URL http://en.wikipedia.org/w/index.php?title=Travelling_salesman_problem&oldid=631582075.

Acknowledgements

Ringrazio la prof.ssa Emanuala Merelli, relatore di questa tesi, per la sua cortesia ed i preziosi consigli ricevuti sia in fase di tesi che durante le lezioni.

Un grazie speciale va alla persona che mi ha fatto appassionare al lavoro che ho svolto e che spero di portare avanti in futuro: il dott. Matteo Rucco. Grazie per avermi spronato a dare il massimo.

Come non menzionare Jacopo Binchi, oltre che correlatore è prima di tutto un amico. I suoi rimproveri e la sua sincerità mi hanno fatto crescere a livello personale e lavorativo.

Ringrazio i miei amici e coinquilini Alessandro, Alessia e Sara con i quali ho condiviso momenti di studio e di dolce far niente.

Infine un grazie particolare va a Massimo l'amico fidato che quando serve c'è sempre, grazie di cuore AMICO MIO.