An algorithm for the detection of cycles in directed graphs

Lorenzo Cioni

Computer Science Department, University of Pisa

e-mail: lcioni@di.unipi.it

Abstract

The present paper contains the description of an algorithm that has been devised for the determination of the cycles in a (even complete) directed graph (**digraph** for short). Given a digraph $G = (N, E)$ the algorithm maps it on a string and then process the string to find within it substrings that satisfy certain properties. Each substring corresponds to a cycle within the given digraph. The paper opens with a short introduction of the problem, then we introduce the algorithm and describe it using a pseudo-code. In the closing section we discuss in short issues of computational complexity.

1 Introduction

The present paper contains the description of an algorithm for the detection and listing of the cycles that are present in a digraph ([Cio03], [Cio06]). Given a digraph $G = (N, E)$, as a set of nodes N and a set of directed arcs or edges E, the algorithm maps it on a string $\sigma \in \Sigma$ whose elements are the identifiers of the nodes that in pairs identify the arcs (and form the alphabet $\mathscr{A}$) and then process the string to identify substrings that satisfy certain properties. To each substring corresponds a cycle in the original digraph. Each of these cycles is identified by listing the identifiers of the nodes of the composing arcs.
If $\sigma \in \Sigma$ is the string corresponding to a given digraph G, the detection of cycles turns, therefore, into the detection of both **connected** and **closed** or **cyclical** substrings i. e. strings[1] $\sigma^* = \sigma_1 \dots \sigma_n = a_1 \dots a_n \subseteq \sigma$ such that:

[1] σ^* is a string composed of n elements $\sigma_i = a_i$, each element is a substring of length 1 and a member of $\mathscr{A}$ and corresponds to a node identifier. We suppose that $\mathscr{A}$ contains

1. $n = l(\sigma^*) > 3$ so that any possible cycle contains at least two arcs[2];
2. $a_i == a_{i+1}$ for $i = 2, 4, \ldots, n-2$ (connectedness);
3. $a_1 == a_n$ (closeness);
4. for $i = 1 \ldots n$ we have[3] $\#a_i = 2$.

The last condition assures that substrings do not contain any cyclical substring and prevents the presence of loops[4]. After having identified all the cyclical substrings, the algorithm discards all the substrings that are equivalent among themselves but one. The equivalence criterion is based on the definition of substrings that are equivalent through shifting. A string:

$$\sigma = a_1 a_2 a_2 a_3 \ldots a_{n-2} a_{n-1} a_{n-1} a_n \tag{1}$$

can be seen, owing to closeness, as a cyclical structure so that a **shift right** of one position (i. e. of one arc) gives us:

$$rs(\sigma, 1) = a_{n-1} a_n a_1 a_2 a_2 a_3 \ldots a_{n-2} a_{n-1} \tag{2}$$

where $a_1 = a_n$. In a dual way we define a **shift left** of one position whereas in a similar way we define a right/left shift of k positions. We say that two strings (and two substrings too) σ^a and σ^b are equivalent through shifting if it exists $k > 1$ such that:

$$\sigma^a = rs(\sigma^b, k) \tag{3}$$

or, equivalently:

$$\sigma^b = ls(\sigma^a, k) \tag{4}$$

Two such strings contain the same arcs but starting from distinct nodes so that they define the same cycle. They are, therefore, equivalent and must be counted as a single cycle.

2 The determination of cycles in directed graphs

A **digraph** $G = (N, E)$ (Figure 1) is characterised by a set of nodes $i \in N$ and a set of oriented arcs $(i, j) \in E$ between pairs of nodes $i,\ j \in N$.

enough symbols to code all the node identifiers of a digraph. If this is not the case we can always find a coding scheme that uses a constant and fixed amount of symbols to code any node identifier so that what follows remain true except for a constant factor.

[2]With $l(\sigma)$ we denote the number of elements of the string σ or its length.

[3]With $\#\sigma_i = \#a_i$ we denote the number of occurrences of σ_i within the string σ.

[4]If a closed path contains only one node is called a loop. A loop represents the reflexive property of the relation $\mathscr{R}$ associated to the arcs of the digraph.

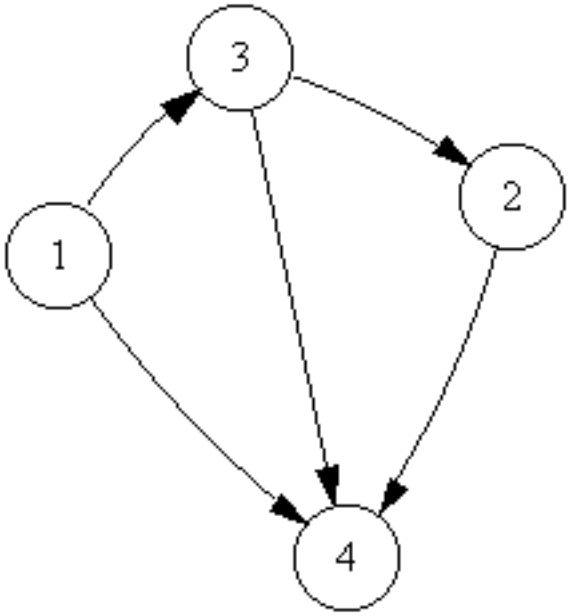

Figure 1: *Example of an acyclic digraph*

A digraph contains a cycle if there is a directed closed path starting and ending on the same node. The cycle is made up of at least two arcs and none of the nodes (but the starting and ending node) is used more than once. In this way we consider only simple cycles. The digraph of Figure 1 does not contain cycles and is termed **acyclic**. If a digraph contains a closed directed path starting and ending on the same node and containing at least two arcs it is said to contain a cycle. The digraph of Figure 2 contains three cycles (described by listing the composing arcs) and precisely:

1. (1,4)(4,1)
2. (1,3)(3,4)(4,1)
3. (1,3)(3,2)(2,4)(4,1)

If we compare figures 1 and 2 we see that the addition of the arc $(4, 1)$ has turned in the definition of the three cycles. In a dual way if we remove one node and the incident arcs[5] we obtain the cancellation of one or more cycles. In Figure 2, if we remove node 1 (and the incident arcs) we get the removal of all the cycles contained in the digraph.

[5]We say that an arc is incident on a node if it has such a node as the source or as destination so that arc (i, j) is incident on the nodes i and j.

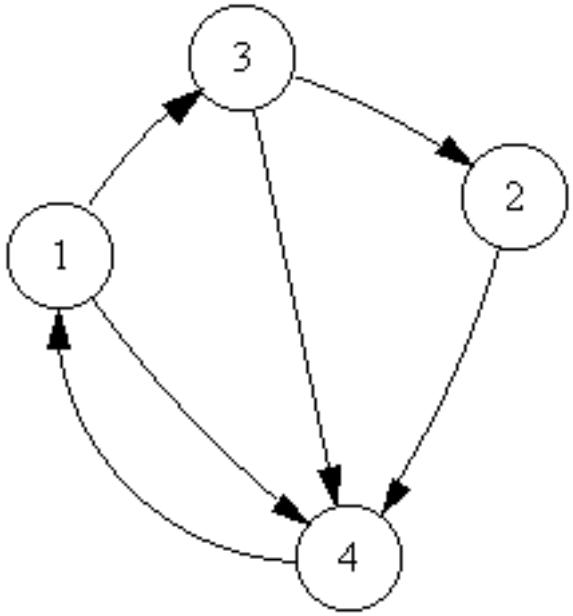

Figure 2: *Example of a digraph with cycles*

3 Digraphs over strings, substrings and cycles

The nodes i of a digraph $G = (N, E)$ are characterised by identifiers id_i belonging to the alphabet $\mathscr{A}$ on whose strings $\sigma \in \Sigma$ the graphs are mapped. Mapping can occur inserting every arc (as a pair of nodes) in the string σ_G corresponding to digraph G. As to the digraph of Figure 2 we can describe it, among the others, in two ways: as a **list of arcs** and an **adjacency matrix**[6].
In the first case, if we use a lexicographic order[7] we get the following array of arcs:

$$(1,3)(1,4)(2,4)(3,2)(3,4)(4,1) \tag{5}$$

[6] An **adjacency matrix** is a square matrix whose dimension is the same as the number of the nodes and whose elements assume one of the values: -1, 1 or 0. Values -1 are only on the main diagonal and identify a node as the source of the arcs with heads corresponding to the 1 on the same row. The 0s means absence of a direct connection between the two row−column nodes.

[7] With **lexicographic order** we mean that we list in increasing order the nodes and, for each node, we list the head of the outgoing arcs in increasing order too.

In the other case we get the square matrix of Table 1. Such a matrix is of easy interpretation. From the representation as the adjacency matrix as

	1	2	3	4
1	-1	0	1	1
2	0	-1	0	1
3	0	1	-1	1
4	1	0	0	-1

Table 1: Adjacency matrix for the digraph of Figure 2

that of Table 1 it is easy to derive a representation such as that given by expression (5). If we denote with A the adjacency matrix we can use the following algorithm[8]:

```
procedure matrixToArray(matrix A)
{
  int size = 0;
  int r = size_of(A);
  int c = r;
  for(int i=1;i<=r;i++)
    for(int j=1;j<=c;j++)
      if (A[i,j]==1) size++; //count the number of arcs
                             //in the digraph
  arcs[] arrayOfArcs := new arcs[size];
  int k=0;
  for(int i=1;i<=r;i++)
     if (A[i,i]==-1)
       for(int j=1;j<=c;j++)
         if (A[i,j]==1)
           arrayOfArcs[k++]:=(i,j);
  return arrayOfArcs;
}
```

Given the representation (5) or of Table 1 it is, therefore, very easy to get a mapping on a string that successively must be processed in search for the cycles of the corresponding digraph. Every cycle corresponds, indeed, to

[8]We use the type *arcs* as a primitive type: an instance of that type is of the form (i, j) with $i, j \in N$. We suppose the type *arcs* is endowed with the necessary primitive operations. We are going to use a pseudo-Java syntax so to make the code more easily readable.

a substring that is both closed and connected (cf. section 1). It is very easy both to modify such an algorithm so to get $matrixToString(matrixA)$ return directly a string (we have only to modify the second pair of nested for loops) and to conceive an algorithm that maps an array of arcs onto a string. In this case we have:

```
procedure arrayToString(arcs[] Ar)
{
  int l = Ar.length;
  String s = "";
  for (int i=0; i < l; i++)
    s = s+toString(Ar[i]);
  return s;
}
```

where $toString()$ is a primitive operation of type $arcs$ that transforms a structure such as (i, j) in the string IJ where I is the symbol of Σ corresponding to identifier i (that may span over more than one digit) and J is the analogous for j. If we apply what we have said to the graph of Figure 2 (as represented e. g. with expression (5)) we obtain the following string:

$$131424323441 \tag{6}$$

that must be processed for the detection of closed and connected substrings.

4 The algorithm

4.1 Introduction

Now we give a description of the algorithm in pseudo-code under the hypothesis that the graph is represented with an adjacency matrix A. The algorithm gets, as input data, a digraph $G = (N, E)$ coded as A and returns an array of strings whose dimension is equal to the number of cycles that are present in G: every string contains the coding of the arcs that form a cycle in G. If it is necessary we can use a decoding procedure so to obtain all the cycles in terms of the composing nodes identifiers.

4.2 The general structure

The proposed algorithm has a structure composed of the **mapping** of a digraph over a string, the **search** for all the closed and connected substrings

and the **removal** of all the duplicate substrings[9].

```
public String[] findLoops(matrix A)
{
    String s=matrixToString(A);
    int lmax=size_of(A)-countEmptyStars(A);
    String[] loops=findClosedStrings(s, lmax);
    loops=removeDuplicates(loops);
    return loops;
}
```

The first step is implemented by a suitable procedure that returns a string corresponding to the digraph G with $m = |E|$ arcs and represented with an adjacency matrix A. The second step defines the maximum length of the cycles whereas the third step ($findClosedStrings(s)$) finds all the connected and closed substrings contained in the string s and returns an array of possibly duplicate strings. The last step ($removeDuplicates(loops)$) remove equivalent through shifting substrings and defines an array of singletons.
Before examining in some detail the third step we make some comments on the second step and so on the maximum length of the cycles $lmax$. If G has n nodes a cycle can contain at the most n arcs. We note, however, that nodes with empty either forward or backward star[10] cannot belong to any cycle so that, if we denote with k the number of such nodes, we have:

$$lmax = n - k \tag{7}$$

It is easy to evaluate k from the adjacency matrix A: k counts the number of rows or columns that contain at the most a -1 on the main diagonal and no other element equal to 1[11]:

```
procedure countEmptyStars(matrix A)
{
  int k = 0;
```

[9] Within the pseudo-code we suppose the primitive types we use (array, matrices, strings and so on) are endowed with the suitable usual primitive procedures. We therefore limit to the description of some unusual and ad hoc procedures.

[10] Given a node identifier i we can identify with the column indexes of the elements equal to 1 of the $i-$th row of the matrix A the elements of its forward star whereas those of its backward star are identified in a similar way but referring to the rows.

[11] We note that there can be a domino effect that we do not investigate in detail since we aim at an algorithm that works well even with complete digraphs. If we find a node i whose forward or backward star is empty we remove both the $i-$th row and the $i-$th column from the matrix A so that it can happen that other nodes get either their forward or their backward star emptied.

```
  int r = size_of(A);
  boolean empty = true;
  for(int i=1;i<=r;i++)
  {
    if(A[i,i]==0) \\FS(i)=emptyset
      k++;
    else
    {
     for(int j=1;j<=r;j++)
      if (A[j,i]==1) empty=false;\\BS(i)!=emptyset
     if(empty)
       k++;
     else
       empty=true;
    }
  }
  return k;
}
```

The value of equation (7) is used within both $findClosedStrings(s)$ and $removeDuplicates(loops)$. In the case of Figure 2, since all the nodes have non empty forward and backward stars, we have that $k = 0$ so that $lmax = n = 4$.
We now examine the procedure $findClosedStrings(s)$ that gets a string as its input and returns an array of connected and closed substrings[12].

```
public String[] findClosedStrings(String s, int lmax)
{
  int l = s.strlength();
  String[] loops = new String[];
  for(int i=2; i <=lmax;i++)
  {
    for(int cur=0; cur<l-1;cur+2)
    {
      FARE
    }
  }
```

[12]The array *loops* is created with no elements and is dynamically increased as needed with a primitive operation $loops.add(s)$, where s denotes a closed and connected string. In a dual way we can remove any string from the array with the operation $loops.remove(pos)$ that removes the element in position *pos*. Operation $s.substring(pos, 2)$ extracts from string s a substring of length 2 starting at the integer position *pos*.

```
}
```

where we have:

```
public boolean checkCloseness{String s}
{
  boolean closed = false;
  int l = s.strlength();
  if(s.charAt(0) == s.charAt(l-1))
    closed = true;
  return closed;
}
```

and:

```
public boolean checkConnectedness{String s1, s2}
{
   boolean connected = false;
   int l = s1.strlength();
   if(s1.charAt(l-1) == s2.charAt(0))
    connected = true;
   return connected;
}
```

The last step is the procedure $removeDuplicates(loops)$ that gets an array of strings and removes from it all the strings equivalent through shifting. The array contains closed and connected strings ordered in increasing length order.

```
public String[] removeDuplicates(String[] loops)
{
}
```

5 Some notes on complexity issues

RIVEDERE!!! We now examine various aspects of the algorithm for what concerns its computational complexity. We suppose to work with a complete digraph G with n nodes and m arcs so that:

$$m = n(n-1) \tag{8}$$

The number of cycles that can be detected in the digraph can be evaluated according to the following considerations. We have to consider cycles of two

arcs and cycles of more that two arcs up to n: the former are counted only once whereas the latter are counted twice because they can be traversed both clockwise and counterclockwise. The number of the cycles that are contained in a complete digraph with n nodes is therefore equal to:

$$\binom{n}{2} + 2\sum_{i=3}^{n}\binom{n}{i} \tag{9}$$

6 Closing remarks and plans for future work

The proposed algorithm represents the implementation of an attempt to analyse complex data structures, such as digraphs, with simpler structures, such as strings, by establishing a one-to-one correspondence between a digraph and a string and between some properties of the digraph with some properties of such a string.
The algorithm must be tuned up, improved as to computational complexity and extended so to allow the detection of other topological properties of the digraphs than the presence and the counting of cycles.

References

[Cio03] Lorenzo Cioni. *Implementazione in Java di strumenti per la simulazione di sistemi dinamici.* Master Degree Thesis "Scienze dell'Informazione", 2003.

[Cio06] Lorenzo Cioni. Graphs and trees: cycles detection and stream segmentation. *Oral presentation AIRO 2006*, Cesena 12-15 September 2006.