# The modified deep first search algorithm: functional implementation

Constantin Ciubotaru

**Abstract**

The article includes the modified deep first search algorithm (DFS) that allows, at a single traversal of a graph, to check its connectivity/biconnectivity, highlight the cut vertices, and build the spanning tree, the biconnected components, and the fundamental set of cycles.

The proposed algorithm was implemented and tested in a functional style using COMMON LISP language. [1]

**Keywords:** undirected and biconnected graphs, spanning tree, $DFS$, cut vertices, fundamental cycles, functional programming.

**MSC 2020:** 68R10, 05C85.

## 1 Introduction

Depth-first search (DFS) is an algorithm for depth traversal in a tree or graph [1], [2]. Initially, the depth traversal method and spanning trees were proposed by the French mathematician Charles Pierre Trémaux (19th century) to solve the labyrinth problem.

The algorithm can be used in the implementation of many problems based on the processing of tree structures. For example, the Travelling Salesman problem, the realization of the Ford-Fulkerson algorithm and the backtracking algorithms, the construction of the spanning tree, the highlighting of cut vertices, the detection of cycles, the graph verification of the connectivity/biconnectivity, the solution of some artificial intelligence problems, web-crawling, etc.


doi:10.56415/csjm.v33.07

[1]The project SIBIA 011301 has supported a part of this research.

The algorithm is also frequently used to check the planarity of graphs and graphs drawing. [3]–[6]. The time complexity of $DFS$ is $O(|V| + |E|)$, where $V$ is the number of vertices and $E$ is the number of edges of the graph.

In the following, we present a modified version of the $DFS$ algorithm taking into account the ideas presented in [4], which allow us in a single traversal to test connectivity and biconnectivity of a graph, highlight cut vertices and biconnected components, and build the fundamental set of cycles.

## 2 Preliminary notions

The undirected graph is denoted by $G = (V, E)$, and the adjacency list of any vertex $v$ – by $adj(v)$.

A sequence of vertices with the property that any two consecutive vertices are adjacent is called *path*: $(v_1\, v_2\, v_3 \ldots v_n)$ or $((v_1\, v_2)\, (v_2\, v_3)\, \ldots\, (v_{n-1}\, v_n))$, where: $(v_i, v_{i+1}) \in E$ for $1 \leq i < n$. The empty path is denoted by *nil* or (). A path is called *elementary* (*simple*) if all participating vertices (edges) are distinct.

A path in which the first vertex coincides with the last is called *cycle*. The cycle is *elementary* if it consists only of distinct vertices, excluding the first and last. The minimum length of a cycle is 3. Two cycles are called *independent* if they differ by at least one edge.

A graph is called *connected* if, for any two distinct vertices of it, there is at least one path connecting them. A vertex is called *cut vertex* (*point of articulation*, *critical vertex*) if the subgraph obtained by eliminating the vertex and its incident edges is no longer connected. A *tree* is a connected graph containing no cycles.

The undirected graph $G$ is called *biconnected* if it has no cut vertices. A *biconnected component* is a maximal biconnected subgraph.

The tree $T = (V, E_t)$ is called a *spanning tree* for the graph $G = (V, E)$ if it contains exactly all the vertices of $G$ and is obtained from $G$ by omitting some edges from $E$, $E_t \subseteq E$. Any connected graph can have multiple spanning trees. Otherwise, the graph will be a tree. The omitted edges form the set of *back edges*. Adding one back edge to the spanning tree will generate a unique cycle. This cycle is called the

*fundamental cycle*. The set of all fundamental cycles represents the *fundamental set of cycles*.

# 3 The modified DFS algorithm

Let's first define the required data structures. The variable *stack* is used to collect all bicomponents. The final set of bicomponents is stored in *bicomponents*. For collecting the lists of cut vertex and back edges, the variables *cutvertex* and *backedges* are used. The set *notconnected* at the end of the algorithm will contain the list of all unreachable vertices from the root (if the graph is not connected or $\varnothing$ otherwise). Four additional lists of size $|V|$ are introduced: *tree*, *sptree*, *ord*, and *up*.

When traversing the graph, any vertex $v$ receives the number that corresponds to its visit order, using counter *visit* and list *ord*. The root will be visited first, assigning it a visit order equal to 0. It is clear that any ancestor $u$ of vertex $v$, $(u\,v)$ or $(u\,v_1\,v_2 \ldots v_n\,v)$ has a visit number smaller than $v$, $ord(u)$¡$ord(v)$.

For any vertex $v$, we denote by $tree(v)$=$u$ the unique direct ancestor of the vertex $v$ in the tree. Thus, the final value of the *tree* will correspond to the spanning tree. Incidence lists *sptree* are also built. An important structure presents the list *up*. For each vertex $v$, $up(v)$ will store the minimal visiting order *ord* of the ancestor (closest to the root) that can be accessed from $v$ using at most one back edge.

The proposed algorithm was implemented and tested in a functional style using the COMMON LISP language [7], [8]. The description of the ***MainDFS*** function is presented in Figure 1. The description of the functions used by ***MainDFS*** is inserted in Figure 2.

First, all assignments are performed by calling the ***data-initiation*** function. At this stage, for any vertex $v$, sets $ord(v)$=-1, $up(v)$= -1, $tree(v)$= -1, $sptree(v)$= nil.

If $v$ is a direct descendant of $u$ and $up(v) \geq ord(u)$, then $u$ will be a cut vertex (Figure 4). For the sets *cutvertex*, *notconnected*, and *bicomponents*, the initial value is set to $\varnothing$, and *backededges* is set equal to $E$.

***mainDFS***

```
function mainDFS(V, E, root)
  call data-initiation;
  call dfstree(root);
  for i=0 to |ord|−1 do
      if ord(i) = -1 then notconnected:=notconnected ∪ {i};
  end for
  if |notconnected| > 0 then print "The graph is not
      connected. The unreachable vertices from the
      root = " notconnected;
      return-from DFS-main ;
      else print "The graph is connected."
  end if
  for i=0 to |tree|−1 do j:=tree(i) ;
      if j ≠ i then
        sptree(j) := (i ||sptree(j));
        backendedges:= backendedges\{(i j),(j i)} ;
      end if
  end for
  if |sptree(root)| = 1 then cutvertex:=cutvertex\{root};
  end if
  print "Final cut vertex = " cutvertex;
  print "Backend edges = " backendedges;
  if cutvertex = ∅ then
    print "The graph is not biconnected.";
    else print "The graph is biconnected.
                bicomponents=" bicomponents;
  end if then else
end mainDFS
```

Figure 1. The description of ***mainDFS*** function

***Additional functions for mainDFS***

```
function data- initiation
    for i=0 to |V|−1 do
        ord(i):=-1; up(i):=-1; tree(i):=-1; sptree(i):=nil;
    end for
    ord(root):=0; up(root):=0; tree(root):=root; visit:=0;
    cutvertex:=∅; notconnected:=∅; stack:=nil;
    backendedges:=E ; bicomponents:=∅;
end data- initiation
```

---

```
function dfs- tree(k)
  for all i ∈adj(k) do
   if(tree(k)≠ i)∧(ord(k)> ord(i))  stack:= (k i)||stack; end if
    if ord(i) < 0 then  /* i is not yet visited */
        tree(i):=k;visit := visit + 1; ord(i):=visit;
        up(i):=visit; /* i is visited */
        call dfs- tree(i);
        up(k):=min(up(i), up(k))
        when up(i) ≥ ord(k)
            curtvertex := curtvertex ∪ {k};
            bic := nil;
            call poppush(k, i, bic);
        end when
    else
        print"i is already visited";
        up(k):=min(up(k), ord(i))
    end if then else
  end for all
end dfs- tree
```

---

```
function poppush(k, i, bic)
  bic := stack(0);
  stack:=pop(stack);
  if(k i) = bic(0)  then push bic in bicomponents else
      poppush(k, i, bic)
  end if then else
end poppush
```

Figure 2. Additional functions for ***mainDFS***

Next is called the recursive function ***dfs-tree***(*root*). This function builds the sets *ord*, *up*, *cutvertex*, and *sptree*. It also manages records in the *stack*. At the time of the appearance of a new bicomponent in the top of the stack, the function ***poppush*** is called, which extracts this component and places it in the set of bicomponents.

For a connected graph, any vertex can be considered a root, thus obtaining multiple spanning trees for the same graph.

Finally, the function ***MainDFS*** checks connectivity/biconnectivity of the graph, builds the lists of adjacency *sptree* and the set of back edges *backedges*, and checks if the root can be a cut vertex.

***Example* 1**

```
V=(0 1 2 3 4 5 6 7 8 9 10 11)
E=((0 5)(0 8)(0 9)(1 4)(1 5)(2 3)(2 5)(2 6)(3 5)(3 7)
   (4 5)(5 8)(6 7)(8 9)(9 10)(10 11))
ADJ=((9 8 5)(5 4)(6 5 3)(7 5 2)(5 1)(8 4 3 2 1 0)(7 2)
     (6 3)(9 5 0)(10 8 0)(11 9)(10))
```

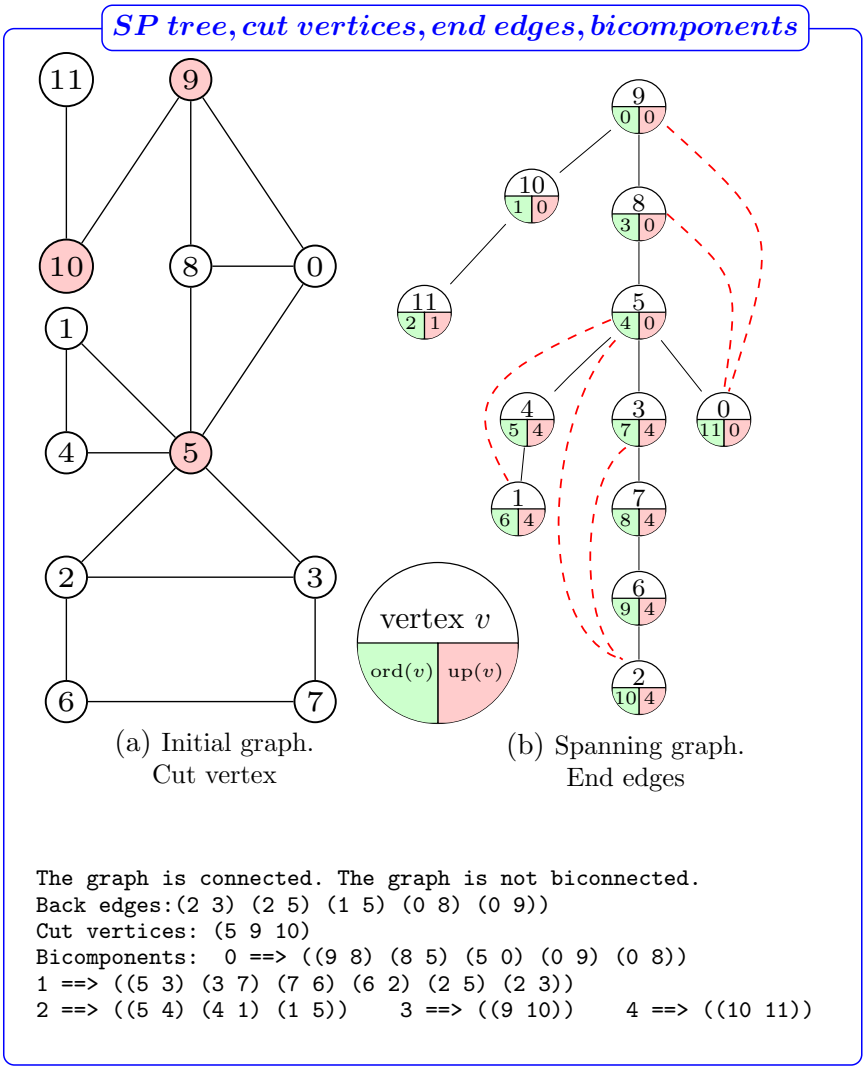
Figure 3. Example 1

An important property of the root should be mentioned. The root of the *sptree* is a cut vertex if and only if $sptree(root) > 1$. The results of performing the ***mainDFS*** function for Example 1 (Figure 3) are presented in Figure 4: (a) – initial graph and cut vertex, (b) – spanning tree, back edges, *ord* and *up* values, bicomponents.

The formal proof of the correctness of the algorithm can be found in [4]. The function ***fundamental-cycles*** is called to calculate the fundamental set of cycles.

## 4 Building the fundamental set of cycles

The fundamental set of cycles is built based on the idea of the Paton algorithm [9] taking into account that the spanning tree *sptree* and the set of back edges *backedges* are already constructed.




(a) Initial graph. Cut vertex

(b) Spanning graph. End edges

Figure 4. The rezults of performing the ***MainDFS*** function

The pseudocodes of the used functions are included in Figure 5. Adding one end edge to the spanning tree generates a unique fundamental cycle.

***PseudocodeFundamental cycles***

```
function fundamental-cycles
  cycles := ∅;
  for u := 0 to |sptree| − 1  do
    call cycles-st(u);
    return cycles;
  end for
end fundamental-cycles
```

---

```
function cycles-st(u)
  stedges := sptree(u);
  if stedges ≠ ∅ then
    for all v ∈ stedges do
      path := (u v); call add-edge-st(path)
    end for all
  end if
end cycles-st
```

---

```
functon add-edge-st
  u := first(path);v := last(path);
  lst := sptree(v); bef :=call backend-edges-from(v);
  if u ∈ bef then
    cicluri := {path ‖u} ∪ cicluri;
    backendedges := backendedges \{(v u)};
  end if
  if lst ≠ ∅ then
    for j := 0 to |lst|− 1  call add-edge-st(path ‖ lst(j))
    end for
  end if
end add-edge-st
```

---

```
function backend-edges-from(v)
  bef := ∅;
  for all (v1 v2) ∈ backendedges do
    if v = v1 then  bef := bef ∪ {v2} ;
    if v = v2 then  bef := bef ∪ {v1} ;
  end for all
  return bef ;
end backend-edges-from
```

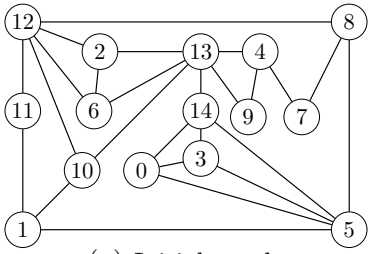
Figure 5. The pseudocode of ***Fundamental-cycles***

***Example 2***

```
V=(0 1 2 3 4 5 6 7 8 9 10 11 12 13 14)
E=((13 14)(1112)(10 13)(10 12)(9 13)(8 12)(7 8)(6 13)
   (6 12)(5 14)(5 8)(4 13)(4 9)(4 7)(3 14)(3 5)
   (2 13)(2 12)(2 6)(1 11)(1 10)(1 5)(0 14)(0 5)(0 3))
ADJ=((3 5 14)(5 10 11)(6 12 13)(0 5 14)(7 9 13)
     (0 1 3 8 14)(2 12 13)(4 8)(5 7 12)(4 13)(1 12 13)
     (1 12)(2 6 8 10 11)(2 4 6 9 10 14)(0 3 5 13))
```

Figure 6. Example 2

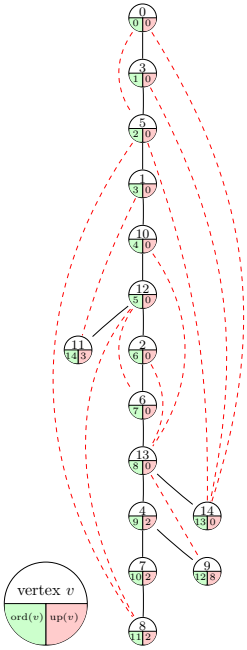
***Example 2. Initial graph, spanning tree***




(a) Initial graph.

The graph is connected.
Cut vertices: $\varnothing$
The graph is biconnected.

```
  The bicomponent:
((0 3)(3 5)(5 0)(5 1)
 (1 10)(10 12)(12 2)(2 6)
 (6 12)(6 13)(13 2)(13 4)
 (4 7)(7 8)(8 5)(8 12)(4 9)
 (9 13)(13 10)(13 14)(14 0)
 (14 3)(14 5)(12 11)(11 1))
```




(b) Spanning tree.
Back edges

Figure 7. The initial graph and spanning tree for Example 2

***Cycles for vertex u = 12 and total cycles.***

```
         Cycles for vertex u=12, cycles-st(12)
(12)=> adj(12)=(11 2)=>(12 11)=> new candidate
(12)=> adj(12)=(11 2)=>(12 2)=> new candidate
(12 11)=>adj(11)={}=>deadlock
(12 11)=>bef(11) = {}=>deadlock , the back edge (11 1) was
                      previously examined and removed
(12 2)  =>adj(2)={6}=>(12 2 6)=> new candidate
(12 2)  =>bef(2)={}=>deadlock
(12 2 6) =>adj(6)={13}=>(12 2 6 13)=> new candidate
(12 2 6) =>bef(6)={12}=>(12 2 6 12)=> new cycle
(12 2 6 13)=>adj(13)={14 4}=>(12 2 6 13 14)=> new candidate
(12 2 6 13)=>adj(13)={14 4}=>(12 2 6 13 4)=> new candidate
(12 2 6 13)=>bef(13)= {9}=> deadlock, the back edges
     (13 2) and (13 10) were previously examined and removed.
(12 2 6 13 14)=>adj(14)={}=> deadlock
(12 2 6 13 14)=>bef(14)={}, the back edges (14 0), (14 3)
                and (14 5) were previously examined and removed.
(12 2 6 13 4)=>adj(4)=(9 7)=>(12 2 6 13 4 9)=> new candidate
(12 2 6 13 4)=>adj(4)=(9 7)=>((12 2 6 13 4 7)=> new candidate
(12 2 6 13 4)=>bef(4)={}=>deadlock
(12 2 6 13 4 9)=>adj(9)={}=>deadlock
(12 2 6 13 4 9)=>bef(9)={13}=>deadlock (13\= 12)
(12 2 6 13 4 7)=>adj(7)={8}=>(12 2 6 13 4 7 8)=> new candidate
(12 2 6 13 4 7)=>bef(7)={}=>deadlock
(12 2 6 13 4 7 8)=>adj(8)={}=>deadlock
(12 2 6 13 4 7 8)=>bef(8)={12}=>(12 2 6 13 4 7 8 12)=>new cycle
     Cycles for vertex u=13, cycles-st(13) ...
     Total fundamental cycles for Example 2:
         0   ====> (13 4 9 13)
         1   ====> (12 2 6 13 4 7 8 12)
         2   ====> (12 2 6 12)
         3   ====> (10 12 2 6 13 10)
         4   ====> (5 1 10 12 2 6 13 4 7 8 5)
         5   ====> (5 1 10 12 2 6 13 14 5)
         6   ====> (3 5 1 10 12 2 6 13 14 3)
         7   ====> (2 6 13 2)
         8   ====> (1 10 12 11 1)
         9   ====> (0 3 5 1 10 12 2 6 13 14 0)
        10   ====> (0 3 5 0))
```

Figure 8. Cycles for vertex u=12 and total cycles

For any back-end edge, a fundamental cycle is built; thus, there will

be obtained a one-to-one correspondence between fundamental cycles and back edges. The algorithm uses the variable *cycles* to store all the fundamental cycles built. For each vertex $u$ of spanning tree, function ***fundamental-cycles*** constructs the set of path $l$=($u\ v_1 \dots v_i\ v$), $i \geq 1$ (checkup list) with generation prospects of cycles. In the case when there exist vertices $u$, for which $sptree(u)$ or/and $sptree(v)$ are empty, this set can not be built. For Example 2 (Figure 6), these are the vertices 4, 6, 7, 8, 9, 11, 14, presented in Figure 7.

The following situations are possible:
1) If $sptree(v) \neq \varnothing$, then for any $z \in sptree(v)$, a new path is generated $l$=($u\ v_1 \dots v_i\ v\ z$), candidate for generating new cycles.
2) If $u \in bef(v)$, then a new cycle is obtained ($u\ v_1 \dots v_i\ v\ u$), which is included in the set *cycles*. In this case, the back edge ($u\ v$) is excluded from *beckedges*, and the path $l$ is excluded from the checkup list.
3). If $sptree(v) = bef(v) = \varnothing$, then $l$ is removed from the checkup list and another candidate path will be examined.

The process will continue as long as there are new chains in checkup list. In Figure 8, it is shown the behavior of the algorithm when generating cycles for vertex 12 of the graph from Example 2.

# 5 Conclusion

The advantage of the algorithm lies in the possibility of verification connectivity/biconnectivity of undirected graphs, the construction of a set of characteristics specific to undirected graphs performing a single traversal of the graph. The functional implementation of the algorithm can be convenient when other applications are developed and could be applied by several users to streamline their activities. Most functions with an intrinsic structure can be used separately. In the future, this algorithm could be used to develop new methods for checking planarity and for drawing graphs.

# References

[1] Shimon Even and Guy Even, *Graph Algorithms*, 2nd ed., Cambridge University Press, 2011, 202 p. ISBN10: 0521517184,

ISBN13: 9780521517188.

[2] A. Gibbons, *Algorithmic Graph Theory*, Cambridge: Cambridge University Press, 1985, 272 p. ISBN: 9780521288811.

[3] H. de Fraysseix, P. Ossona de Mendez, and P. Rosenstiehl, "Trémaux trees and planarity," *International Journal of Foundations of Computer Science,* vol. 17, no. 5, pp. 1017–1029, 2006. DOI: 10.1142/S0129054106004248.

[4] John Hopcroft and Robert Tarjan, "Efficient planarity testing," *Journal of the ACM,* vol. 21, no. 4, pp. 549–568, Oct., 1974. DOI: https://doi.org/10.1145/321850.321852.

[5] Takao Nishizeki and Md. Saidur Rahman, *Planar Graph Drawing* (Lecture Notes Series on Computing, vol. 12), World Scientific, 2004, 312 p. DOI: https://doi.org/10.1142/5648. ISBN: 978-981-256-033-9.

[6] Roberto Tamassia, *Handbook of Graph Drawing and Visualization* (Discrete Mathematics and Its Applications Ser.), Chapman & Hall/CRC, 2016, 862 p. ISBN: 978-1-138-03424-2.

[7] G. L. Steele, *Common Lisp. The Language*, 2nd ed., USA: Thinking Machines, Inc. Digital Press, 1990, 1029 p. ISBN: 1-55558-041-6.

[8] CLISP – an ANSI Common Lisp, Slashdot Media. [Online]. Available: `http://sourceforge.net/projects/clisp/files/clisp/2.49/`. Accessed August 20, 2024.

[9] K. Paton, "An algorithm for finding a fundamental set of cycles of a graph," *Communications of the ACM*, vol. 12, no. 9, pp. 514–518, Sept. 1969. DOI: https://doi.org/10.1145/363219.363232.

Constantin Ciubotaru

Received August 18, 2024
Accepted August 29, 2024

ORCID: `https://orcid.org/0009-0005-8896-0966`
Moldova State University,
Vladimir Andrunachievici Institute of Mathematics and Computer Science
E–mail: `constantin.ciubotaru@math.usm.md, chebotar@gmail.com`