

From the Proseminar  
“Algorithms and Data Structures”  
at Prof. Brandenburg, SS2001

# Graph Traversals

by Franz Lenhardt

## 1 Introduction

When dealing with graphs, one fundamental issue is, to traverse the graph, to go through it. In other words: to visit each vertex and each edge. To solve this problem, many interesting algorithms exist. Two of them will be presented here: Depth-First Search (abbr. dfs) and Breadth-First Search (bfs). These two methods are used e.g. in connection with the task of finding the connected components of a graph, which is a nice example of an application of bfs and dfs.

## 2 Depth-First Search (dfs)

The dfs algorithm will become clear when taking this example: Imagine a single person being trapped inside a maze. In order to get out, the person has to be sure to visit each path and each intersection. So he or she uses two colors of paint, to mark the intersections already passed. When discovering a new intersection, it is marked grey, and the way deeper into the maze is continued. After reaching a “dead end” at the end of each path from an intersection though, the person knows that there is no more unexplored path from the grey intersection, which now is completed and thus can be marked black. This “dead end” is either an intersection which has already been marked grey or black, or simply a path that does not lead to an intersection.

The connection to the issue of graphs is obvious: The intersections of the maze are the vertices, while the paths between the intersections are the edges of the graph. The technical term for the returning of the person from the “dead end” is *backtracking*. So what you are doing, is to go away from your starting vertex into the graph as *deep* as you can, until you have to backtrack to the preceding grey vertex.

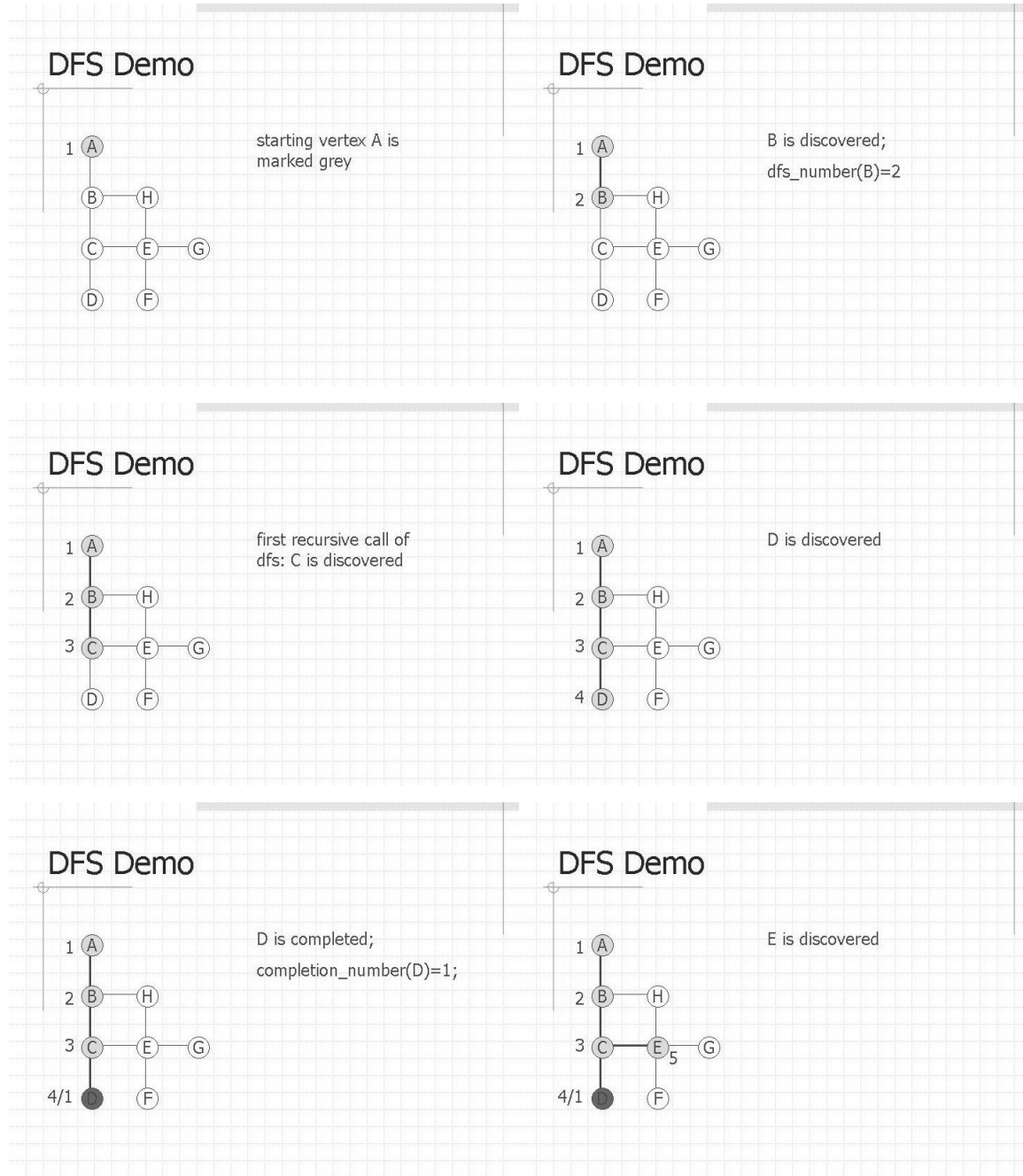
To gain more information from the algorithm, not only a color value (white, grey or black) is assigned to each vertex, but also two numbers: the *dfs\_number* and the *completion\_number*. While the first number is assigned when a vertex is discovered (marked grey), the latter is allocated, when it is completed (marked black).

Here the pseudo-code of the dfs algorithm:

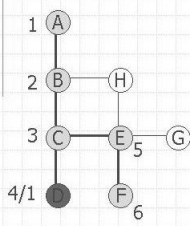
```
i:=1; j:=1; //initialisation of iteration variables
PROCEDURE depth_first_search(u: vertex)
color[u]:=grey;
dfs_number[u]:=i++;
FOR v ∈ Adj(u) DO //for each vertex adjacent to u
    IF color[v]=white THEN depth_first_search(v)
color[u]=black;
completion_number[u]:=j++;
```

As you can see, the algorithm is implemented by the use of recursion. It would also be possible to use a LIFO-Stack, where each newly discovered vertex is pushed into, and popped out, when it is completed.

Now see this example of a dfs traversal of an undirected and unweighted graph:

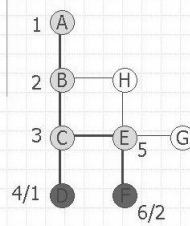


### DFS Demo



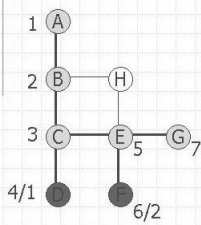
F is discovered

### DFS Demo



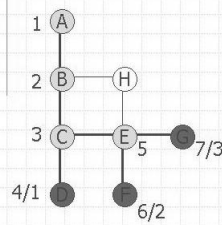
F is completed

### DFS Demo



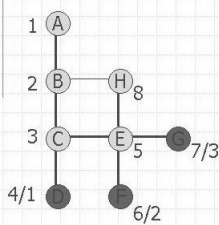
G is discovered

### DFS Demo



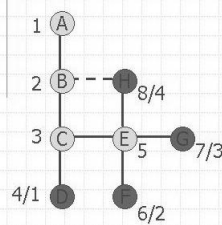
G is completed

### DFS Demo



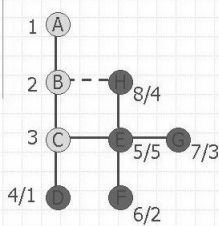
H is discovered

### DFS Demo



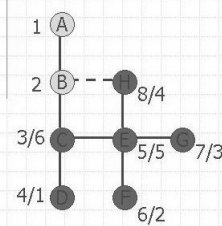
H is completed;  
(B,H) is a **back edge**

### DFS Demo

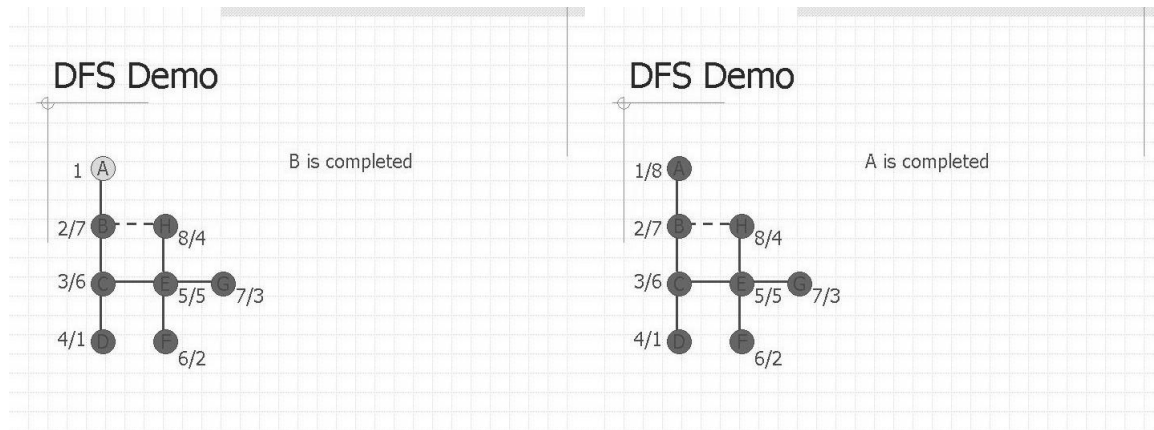


E is completed

### DFS Demo



C is completed



Here you could see that sometimes an edge leads to an already discovered vertex. These edges are called *back edges*, while the other edges are called *tree edges*. They are called like that because a tree can be made of such a graph by seeing the tree edges as the branches of the tree, while the back edges are deleted. And this is already a task to be solved by dfs: to construct a tree from a graph, where the root of the tree is the starting vertex.

When viewing the example, another property of dfs comes clear: The discovery time and completion time have parenthesis structure. This means, the vertex discovered first is the last one to be completed. The reason for this can also be seen at the recursion, where dfs is started on a vertex until it has to track back, i.e. the recursion unrolls.

Finally, the running time of dfs is  $O(|v|+|e|)$ , as each vertex and each edge have to be visited.

Dfs is used to check whether there is a path between two given vertices, and if it is, to compute this path. As stated above, a spanning tree can be constructed by using dfs, and also the connected components of a graph, which will be the last paragraph of this report.

### 3 Breadth-First Search (bfs)

While in the dfs example the person was trapped alone inside the maze, in this case there are several people “swarming out” from a starting point. At first, they move away to each neighbour of the starting point, then on to the neighbours of the neighbours and so on. By that way, they are exploring the maze by “waves”, as if they were unrolling a string from the starting point to each neighbour intersection at each wave.

And that is the most important property of bfs: It yields the shortest path in terms of the number of edges between any vertex and the starting vertex, as each vertex is assigned a number: It is called `bfs_number` and tells us about in which level from the starting point the particular vertex lies in. This makes bfs similar to the Dijkstra algorithm, if we explore a graph where each edge has the same weight.

The pseudo-code of bfs uses a queue  $Q$  to save the newly discovered vertices, whose neighbours have not been examined yet.

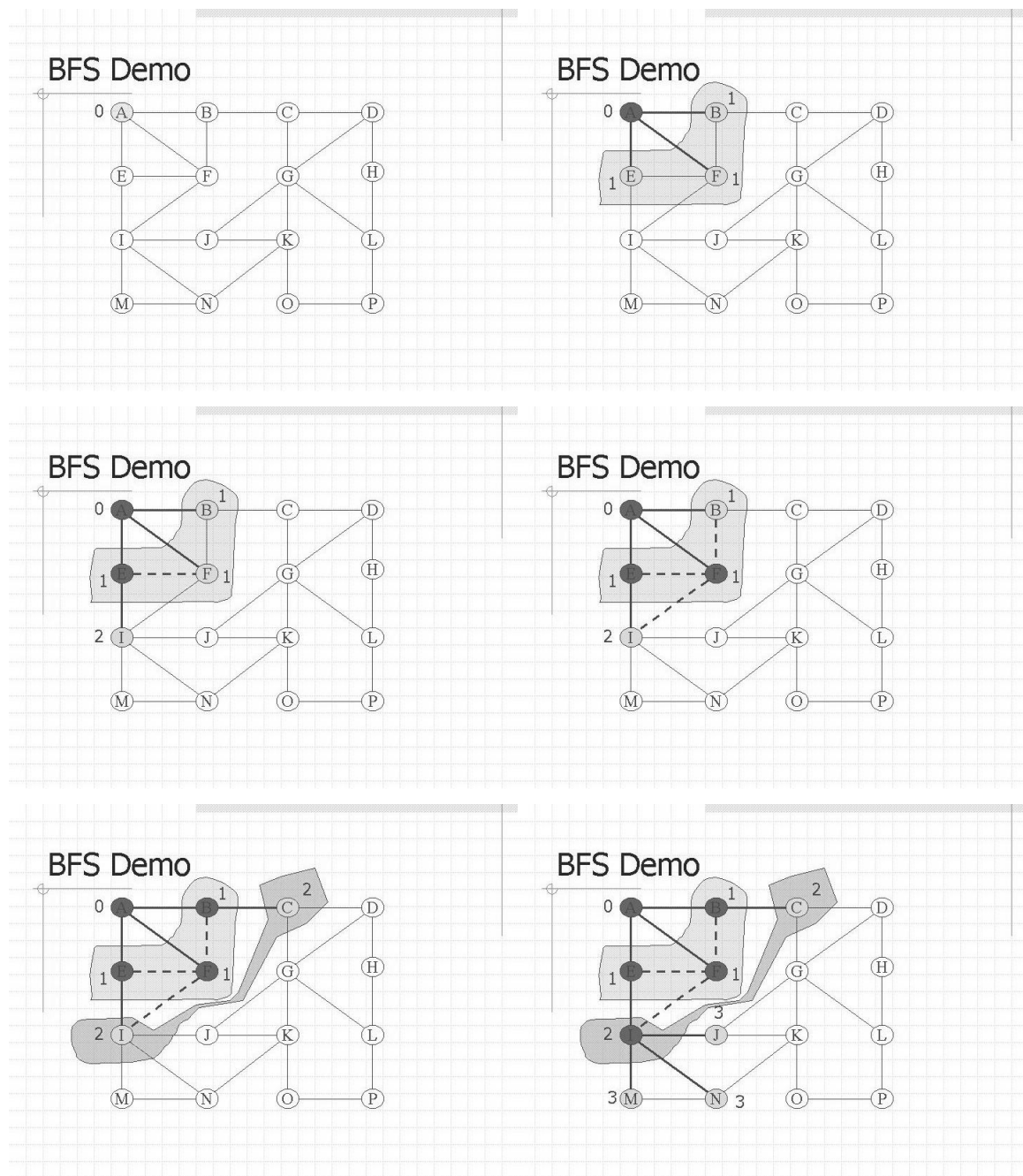
```

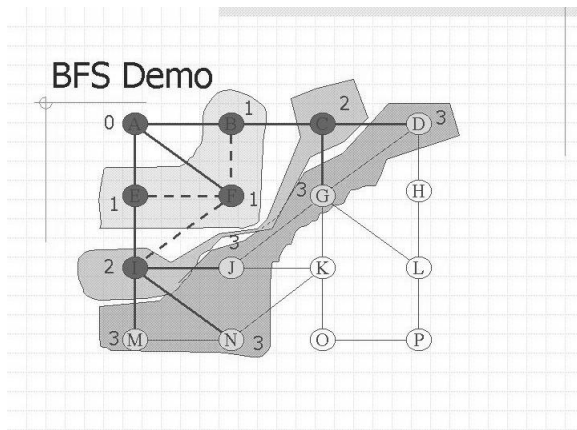
FOR  $v \in V$  DO
   $d[v] := 0$ 
   $Q := \{v\}$ 
  WHILE  $Q \neq \emptyset$  DO
     $u := \text{hd}(Q)$ 
     $Q := \text{tl}(Q)$ 
    FOR  $v \in \text{Adj}(u)$  DO
      IF  $\text{bfs\_number}[v] = \text{nil}$  THEN
         $\text{bfs\_number}[v] := \text{bfs\_number}[u] + 1$ 
         $Q := Q \cup \{v\}$ 

```

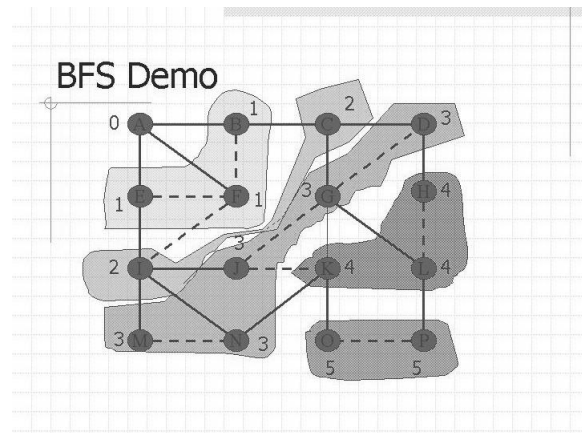
//as long as the queue is not empty,  
 //take the first element  $u$  of  $Q$   
 //delete it from  $Q$   
 //and check each neighbour of  $u$   
 //if it is still undiscovered  
 //assign it a number  
 //and put it into  $Q$

Here an illustration of a bfs traversal (again on a non-directed unweighted graph):





And so on, until the whole graph is examined and every vertex is assigned a number:

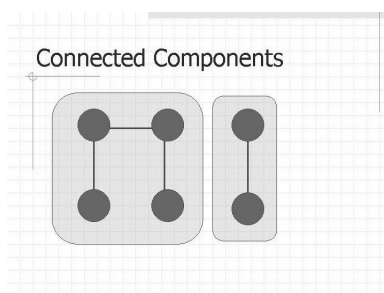


The edges of the graph are classified as *tree edges* and as *cross edges*. While the *tree edges* span a tree from the graph (just like the tree edges of dfs traversal), the *cross edges* are connections of vertices being in the same level or inside two neighbouring levels (i.e. with their bfs\_number differing at most by 1).

As with dfs, the running time of bfs is of  $O(|E|+|V|)$ .

Bfs traversal can not only be used to produce a tree from a graph, but also to compute the connected components of a graph.

#### 4 Connected Components



Before constructing a tree from a graph via dfs or bfs, it is suitable to know whether the given vertices lie inside the same connected component of the graph.

Therefore, dfs or bfs traversal can be used: The program starts by choosing an arbitrary vertex of as starting point of the bfs/dfs algorithm. After it has completed, we check if each

vertex of  $G$  has already been visited. If not, we know that there are at least two connected components in the graph. So we start dfs/bfs on one of the remaining unvisited vertices as long until each vertex has been examined.

The input of such a program would be a graph, while the output could either be several lists of the vertices within the same connected component or only one vertex of each component as a representation of the component.

## 5 Used Literature

- T. Ottmann; P. Widmayerr: Algorithmen und Datenstrukturen, Spektrum Verlag Heidelberg, S. 551ff
- M. T. Goodrich, R. Tamassia: Data Structures and Algorithms in Java, John Wiley and Sons, Inc., 1998, §9.3, §9.4.1
- T. H. Cormen; C. E. Leiserson; R. L. Rivest: Introduction to Algorithms, The MIT Press, Cambridge, Mass., p.488ff