Graphs: Definitions and Basic Algorithms

Lluís Alsedà
Revised and improved by Albert Ruiz

Departament de Matemàtiques
Universitat Autònoma de Barcelona
`http://www.mat.uab.cat/~alseda`

Version 3.54 (April, 2023)

UAB Universitat Autònoma de Barcelona DEPARTAMENT DE **MATEMÀTIQUES**

Table of Contents

Graphs and Trees — Introduction

Contents

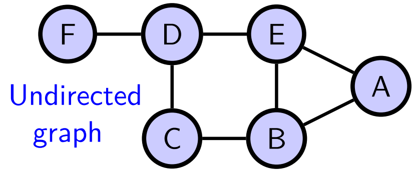
1. Introduction
2. A bit of history

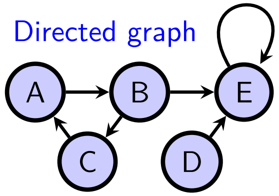
Graphs and Trees — Introduction[1]

A *combinatorial graph* is an ordered pair $G = (V, E)$ of *vertices* or *nodes* V, and a subset $E \subset V \times V$ of the Cartesian product $V \times V$.

In the case of an *undirected* graph, the elements of E are called *edges* and the pairs $(v, w) \in E$ are considered without ordering (that is, there is an edge between $v \in V$ and $w \in V$ when $(v, w) \in E$ or $(w, v) \in E$).

In the case of a *directed* or *oriented* graph, the elements of E are called *arrows* and the pairs $(v, w) \in E$ are considered with ordering (that is, there is an arrow from $v \in V$ to $w \in V$ if and only if $(v, w) \in E$).



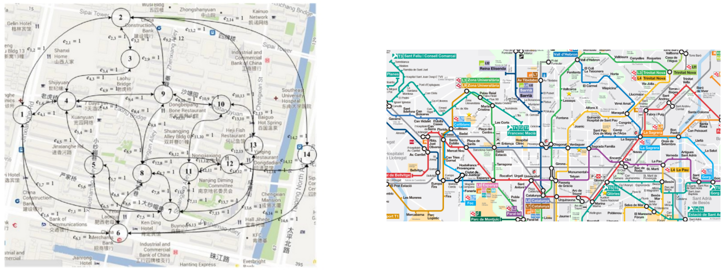


---

[1] `http://en.wikipedia.org/wiki/Graph_theory`

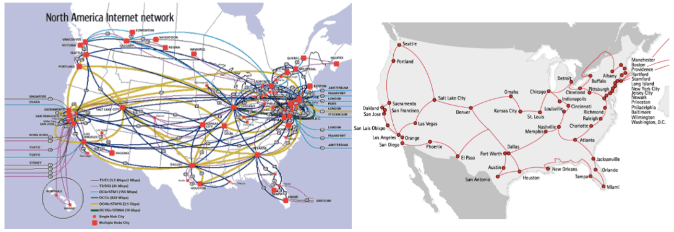
Graphs and Trees — Introduction

Graphs are used to represent communication networks, data organizations, computing devices, computing flows and, currently, in all disciplines from linguistics to sociology and biology, to mention a highly restricted list of examples.



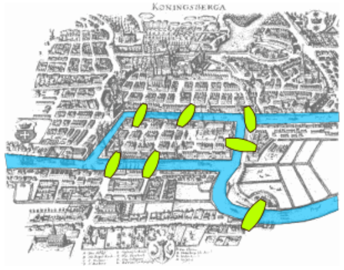
Graphs and Trees — Introduction

For example, the link structure of a website can be represented and studied by means of a directed graph, in which the vertices represent web pages and directed edges represent links from one page to another.



A little bit of history

The article written by Leonhard Euler on the seven bridges of Königsberg and published in 1736 is considered the first document in the history of graph theory. In this work, as well the one written by Vandermonde on the knight problem, they studied what today is known as the *Euler Formula* relating the number of edges, vertices, and faces of a convex polyhedron, that is at the origin of the topology.



---

[2] Figure extracted from Wikipedia

Graphs and Trees — Introduction

One of the most famous and stimulating problems in graph theory is, and it has been the problem of the four colors:

It is true that any map drawn in the plane can have its regions coloured with four colors, such that two regions who have a common border have different colors?

Graphs and Trees — Basic Definitions

Contents

Basic definitions — Order and Size

Order of a graph

The *order* *of a graph is the number of vertices* $|V|$.

Example: *The graphs on page 2 have order 6 and 5 respectively.*

Size of a graph

The *size* *of a graph is the number of edges or arrows* $|E|$.

Example: *The graphs on page 2 have size 7 and 6 respectively.*

Basic definitions — Valence and Degree

Degree of a vertex

The *degree* *or* *valence* *is the number of edges reaching or leaving the vertex. If an edge connects a vertex with itself it counts twice.*

Example: *The vertex* D *of the undirected graph of page 2 has valence 3 while vertex* E *of the directed graph of the same page has valence 4.*

Degree of a vertex — Directed case

The *in-degree* *is the number of edges that reach the vertex.*
The *out-degree* *is the number of edges leaving the vertex.*

Example: *The vertex* E *of the directed graph of page 2 has in-degree 3 and out-degree 1.*

Basic definitions — Leaf and Branching vertices

Leaf vertex

The vertices belonging to a single edge (i.e. the vertices of valence 1) are called *terminal* *or* *leaf.*

Example: *The only leaf vertex of the undirected graph of page 2 is vertex* F*, and the only leaf of the directed graph of the same page is vertex* D*.*

Branching vertex

A *branching* *vertex is any vertex with valence greater than two.*

Example: *The branching vertices of the undirected graph of page 2 are* B*,* D *and* E*, and the branching vertices of the directed graph of the same page are* B *and* E*.*

Basic definitions — Paths and loops

Path

A path *is a sequence of edges connected linearly. If the graph is directed the end of an arrow should be the beginning of the next one.*
The length *of a path is its number of edges.*

Example: (F, D, C, B, A) *is a path length 4 of the undirected graph from page 2, while* $B \to C \to A \to B \to E \to E \to E$ *is a path of length 6 of the oriented graph of the same page.*

Loop or Circuit

A loop *or* circuit *is a closed path. That is, the end of the last edge is the beginning of the first one.*

Example: (B, C, D, E, B) *is a length 4 loop of the undirected graph in page 2, while* $B \to C \to A \to B$ *is a length 3 loop of the oriented graph of the same page.*

Connectedness in graphs

Contents

Connectedness in undirected graphs

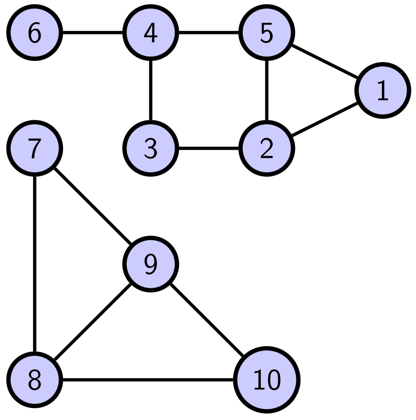
Connectedness

An undirected graph is connected *when there is a path between each pair of vertices (i.e., there are no inaccessible vertices).*

Connected Component

A connected component *of an undirected graph is a maximal connected subgraph.*
Note that each vertex and each edge belongs to a single connected component.

Example: An unconnected graph with two connected components.

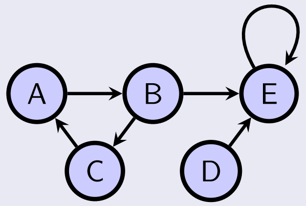


Connectedness in directed graphs

Weak connection

A directed graph is called weakly connected *when it is connected as an undirected graph. That is, when replacing all its (directed) arrows with undirected edges we get a connected (undirected) graph.*

Example:
The directed graph of page 2



Connectedness in directed graphs

Semi-connection

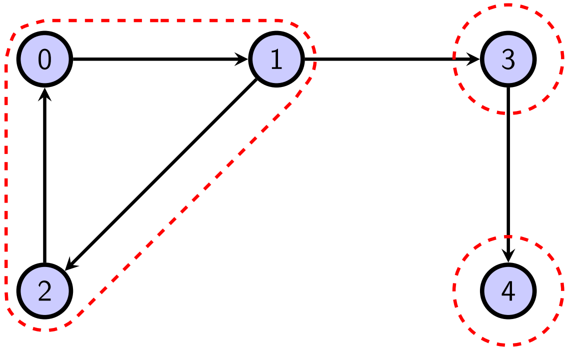
A directed graph is called unilaterally connected or semi-connected when, given any two vertices u, v, it contains a path from u to v or a path from v to u.

Strong connection

A directed graph is called strongly connected when, given any two vertices u, v, it contains a path from u to v and a path from v to u. A strongly connected component is a strongly connected maximal subgraph.

Strongly connected components

Example: a directed graph which is not strongly connected, with three strongly connected components.



Memory models

Contents

1. List representations — Adjacency List
2. List representations — Incidence List
3. Matrix representations — Adjacency Matrix
4. Matrix representations — Incidence Matrix

Memory models

There are different ways to store graphs in memory.

The used data structure depends on the structure of the graph and also on the algorithm used to manipulate the graph. There are two basic types of representations: lists and matrix structures.

For scattered graphs (with few edges) the representation as a list structure is often preferred as it has smaller memory requirements.

List representations — Adjacency List

Adjacency list

The vertices are stored as structures, and each vertex stores a list of adjacent vertices. This data structure allows the storage of additional data about vertices (e.g. latitude and longitude in the case of geographic data).

An example in C— The undirected graph of page 2

Made with vectors of fixed size.
It is more inefficient but simpler.

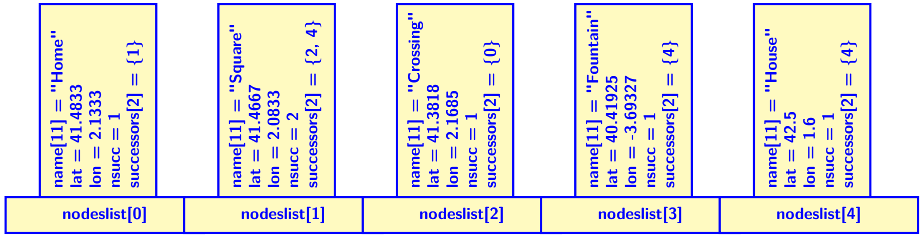
```
typedef struct {
  char name;
  unsigned short nsucc;
  unsigned short successors[3];
} node_simple;
node_simple GrafNO[6]={{’A’, 2, {1, 4}},
                       {’B’, 3, {0, 2, 4}},
                       {’C’, 2, {1, 3}},
                       {’D’, 3, {2, 4, 5}},
                       {’E’, 3, {0, 1, 3}},
                       {’F’, 1, {3}}   };
```

List representations — Adjacency List

Another example: the directed graph of page 2

Made with vectors of fixed size.
It is more inefficient but simpler.

```
typedef struct {
  char name[11];
  double lat, lon;
  unsigned short nsucc;
  unsigned short successors[2];
} node;
node nodeslist[5] = {{"Home",     41.4833,   2.1333,  1, {1}},
                     {"Square",   41.4667,   2.0833,  2, {2, 4}},
                     {"Crossing", 41.3818,   2.1685,  1, {0}},
                     {"Fountain", 40.41925, -3.69327, 1, {4}},
                     {"House",    42.5,      1.6,     1, {4}}};
```



List representations — Incidence List

Incidence list

Vertices and edges are stored as structures. Each edge stores its incident vertices. In addition, optionally, each vertex can store its incident edges. This data structure allows the storage of additional data on vertices and edges (e.g. names, weights, . . .).

An example in C— The directed graph of page 2

```
typedef struct {
  char name [11];
  double lat, lon;
} node;

node llnod [5] = {
   {"Home", 41.4833, 2.1333},
   {"Square", 41.4667, 2.0833},
   {"Crossing", 41.3818, 2.1685},
   {"Fountain", 40.41925, -3.69327},
   {"House", 42.5, 1.6}};
```

```
typedef struct {
  unsigned short begin;
  unsigned short end;
} dir_edge;

unsigned short graph_size = 6;

dir_edge edges [graph_size] = {
      {0, 1},{1, 2},{2, 0},
      {1, 4},{3, 4},{4, 4}
};
```

Matrix representations — Adjacency Matrix

Adjacency matrix

It is an array (two-dimensional — with two indexes), in which the rows represent the starting vertices and the columns represent the final ones. The entry i, j in the array stores the number of arrows that start at i and end at j. In an undirected graph this matrix is symmetric. Additional data on edges and vertices must be stored apart.

Again the directed graph of page 2

$$\begin{pmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

Matrix representations — Incidence Matrix

Incidence matrix

It is a two-dimensional boolean matrix, in which the rows represent the vertices and the columns represent the edges. Its entries indicate if the vertex in a row is incident at the edge of a column.
For directed graphs
$+1$ indicates that the vertex is the origin of the edge, and
-1 indicates that the vertex is the end of the edge.

Again the directed graph of page 2

The edges are numbered as follows:
$\alpha_1 = 1 \to 2,\ \alpha_2 = 2 \to 3,\ \alpha_3 = 3 \to 1,\ \alpha_4 = 2 \to 5,\ \alpha_5 = 4 \to 5.$

$$\begin{pmatrix} 1 & 0 & -1 & 0 & 0 \\ -1 & 1 & 0 & 1 & 0 \\ 0 & -1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & -1 & -1 \end{pmatrix}$$

Note that this graph cannot be represented in this way: Indeed, the edge $5 \to 5$ cannot be represented.

Graph traversal — Introduction

Unlike vectors, linked lists, and other linear data structures, which can be traversed canonically in linear order, trees and graphs can be traversed in several ways which are essentially different since *they are not linearly ordered*.

Traversing a graph involves iteration over all nodes and, given a fixed a node, there is more than one possible choice for the *next node* (we are not in a linear data structure). Therefore, in a sequential computation framework, algorithms and rules are needed to determine in each case the choice of the *next node*. Moreover these algorithms and rules have to regulate the treatment of the other (remaining) adjacent nodes, to visit them later.

There are basically two philosophies of graph traversal:

- *In depth*: When choosing the next node we prioritize to *increase* the depth.
- *By levels*: When choosing the next node we prioritize to *maintain* the depth level.

Graph traversal — Introduction

Since a graph is a self-referential (recursively defined) data structure, the traversal can be implemented very naturally and clearly by recursivity (in this case, deferred nodes are implicitly stored in the stack).

Observations/problems of the *depth* notion

- It is not an absolute notion. It clearly depends on what the *source node* is.
- As we will see, the definition is simple and canonical (independent of the path taken) when there is a unique path between the source node and each of the other nodes.

- The first problem is easy to solve: just specify the source node.
- The second problem is more difficult to solve. We will do this in two stages: initially we will deal with the easy case of *trees* (for which, *given two nodes, there is a single path that connects them*). Then it will be easier to deal with the general case of an arbitrary graph.

Trees and Rooted trees

Contents

Trees [2] — Uniquely arcwise connected

Definition

- *A tree is a connected graph without loops (circuits).*

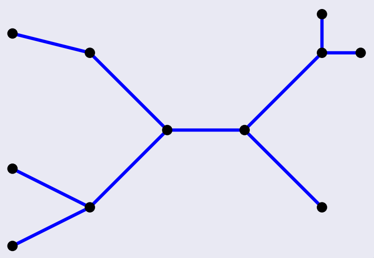
Equivalently:

- *A tree is a uniquely arcwise connected graph: Any two vertices are connected by a single path.*

Properties

- Adding an arbitrary edge to a (non oriented) tree forms a loop.
- Deleting any edge the tree gets disconnected.
- A tree with n vertices has exactly $n-1$ edges (the *Euler characteristic* is equal to 1).

Example: a tree



---

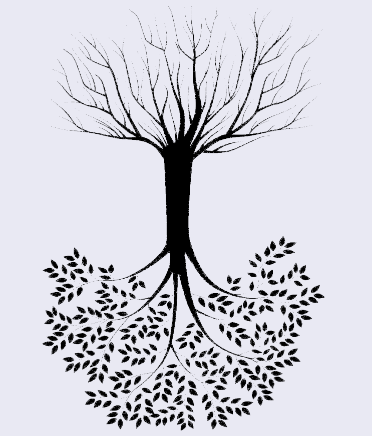
[2] `http://en.wikipedia.org/wiki/Tree_(graph_theory)`

Rooted trees: Fixing the source node

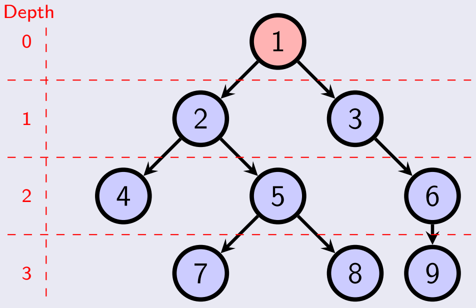
Rooted trees

A rooted tree is a tree in which one vertex has been designated to be the root or source node.

Example: A rooted tree



Example — Another rooted tree: With **1** as root or source node



Example: Vertices 4, 7, 8, and 9 are *leaves or end-nodes*.
Vertices 2 and 5 are *branching nodes*.

Rooted trees: Leaf and Branching vertices

Leaf and Branching vertices

For rooted trees, the *leaf* and *branching* vertices are usual *leaf* or *branching* vertices *which are different from the root.*

Example: the rooted tree from the previous page

The leaves are the vertices 4, 7, 8 and 9.
The vertices 2 and 5 are branching.

Note: *The valence of a vertex does not depend on the root node*

Therefore, *the leaves and branching vertices of a tree are independent of the root node except for the root node itself* (which abandons its character when designated root).

Rooted trees — Depth of a vertex

Definition: Depth of a vertex

In a rooted tree the depth of a vertex is defined as the distance from this node to the root.

Distance: The distance between two nodes is measured as the length of the unique path that connects them (remember that a tree is uniquely arcwise connected).
Obviously the distance from a node to itself is 0.

Note: The root depth is 0.

Example: Depths of the rooted tree of page 28:

Depth	**0**	**1**	**2**	**3**
Vertices	1	2	4	7
		3	5	8
			6	9

Example: The depth of the node $v = 5$

$1 \longrightarrow 2 \longrightarrow 5$ is the only path that connects the root with node 5, and this path that has length 2.

Rooted trees — Depth of a vertex

Important Note (to remember)

The depths of the vertices of a rooted tree depend on the root.

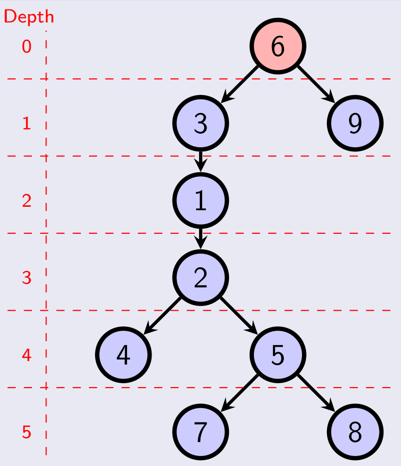
Depths of the example on the right (compare with the example above)

Depth	**0**	**1**	**2**	**3**	**4**	**5**
Vertices	6	3 9	1	2	4 5	7 8

Remarks:
When we switch the root from node 1 to 6:

- the only vertex that keeps the same depth is the node 3.
- leaves and terminal vertices do not vary (since both roots have valence 2).

Example: The tree in page 28 with vertex 6 as root



Rooted trees
Depth as vertex ordering
Note (explaining a previous comment)

A rooted tree defines a partial ordering in the set of vertices in the direction of increasing depth
(see the rooted trees on pages 28 and 31).

The root is the smallest vertex, and the leaves are the maximum ones.

Rooted trees — Tree depth

Definition

The depth of a rooted tree is defined as the maximum depth of the vertices (as above, it depends on the chosen root).

Example

- The rooted tree 1 on page 28 has depth 3.
- The same root tree 6 on page 31 has depth 5.

Rooted Trees — Parents and Children

Definition: *parent*

Given a rooted tree and a vertex v of depth $p > 0$ (i.e. v is not the root), the *parent of v* is defined as the unique vertex adjacent to v of depth $p-1$. Equivalently, the *parent of v* is the node adjacent to v in the unique path that connects the root with v.

Obviously the root has no parent (in fact it is the only vertex that has no parent).

Example: Parents of the tree in page 28

vertices	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
even	-	1	1	2	2	3	5	5	6

Example: Parents of the tree in page 31

vertices	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
even	3	1	6	2	2	-	5	5	6

Rooted Trees — Parents and Children

Definition: *child*

Let v be a vertex of depth $p \geq 0$ in a rooted tree. A node with depth $p+1$ adjacent to v is called a *child of v*.

- Leaves have no children (and are the only vertices that have no children).
- A vertex which is not a leaf can have more than one child.

Example: Children of the tree in Page 28

vertices	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
children	2	4	6	–	7	9	–	–	–
	3	5			8				

Exemple: Children of the tree in Page 31

vertices	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
children	2	4	1	–	7	3	–	–	–
		5			8	9			

n-ary trees

Definition

An n-ary tree is a rooted tree for which each vertex has as up to n children.

- *2-ary trees are called binary, and*
- *3-ary trees are called ternary.*

Example

The rooted trees in pages 28 and 31 are binary trees.

Tree traversal

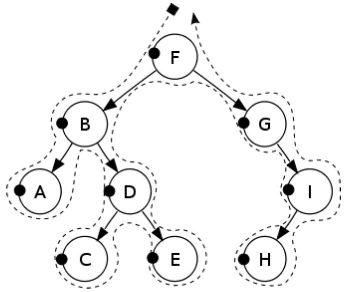
Contents

1. In depth: *depth-first search* — pre-order
2. In depth: *depth-first search* — in-order
3. In depth: *depth-first search* — post-order
4. By levels: *breadth-first search*

Tree traversal[3] in depth: *depth-first search* `pre-order`

Description

This search algorithm first visits (lists) every traversed node. Then, it moves towards the left child and, upon the second traversal of the node (uphill), it moves towards the right child.



Visiting order:
F, B, A, D, C, E, G, I, H

---

[3] `https://en.wikipedia.org/wiki/Tree_traversal`

Tree traversal in depth: *depth-first search*
pre-order — pseudocode

Algorithm: recursive pre-order

```
procedure PREORDER(node)
    if (node = null) then
        return
    end if
    visit(node)
    preorder(node.left)
    preorder(node.right)
end procedure
```

Comment

The initial check (**if**) is necessary to detect when we are on a leaf and, in this case, stop the recursive algorithm because we can't go deeper.

Algorithm: iterative pre-order

```
procedure ITERATIVEPREORDER(noderoot)
    s ← EmptyStack
    s.push(noderoot)
    while (not s.IsEmpty) do
        node ← s.pop
        visit(node)
        if (node.right $\neq$ null) then
            s.push(node.right)
        end if
        if (node.left $\neq$ null) then
            s.push(node.left)
        end if
    end while
end procedure
```

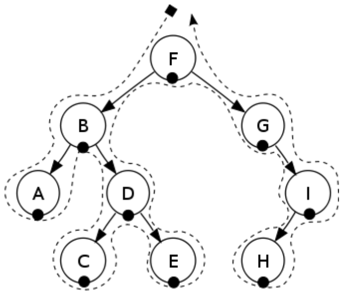
Comment

It is understood that we do not cheat by entering noderoot **null**.

Tree traversal in depth: *depth-first search*
in-order

Description

This search algorithm first moves towards the left child of every traversed node. Afterwards, upon the second traversal of the node (uphill), it visits the node and moves to the right child.



Visiting order:
A, B, C, D, E, F, G, H, I

Tree traversal in depth: *depth-first search*
in-order — pseudocode

Algorithm: recursive in-order

```
procedure INORDER(node)
    if (node = null) then
        return
    end if
    inorder(node.left)
    visit(node)
    inorder(node.right)
end procedure
```

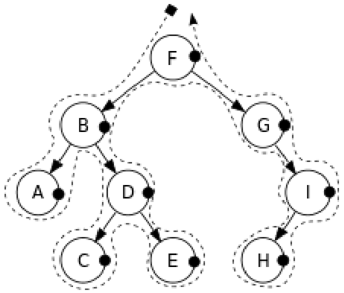
Algorithm: iterative in-order

```
procedure ITERATIVEINORDER(noderoot)
    s ← EmptyStack
    node ← noderoot
    while (true) do
        if (node $\neq$ null) then
            s.push(node)
            node ← node.left
        else
            if (s.IsEmpty) then return; end if
            node ← s.pop
            visit(node)
            node ← node.right
        end if
    end while
end procedure
```

Tree traversal in depth: *depth-first search*
post-order

Description

This search algorithm first moves towards the left child of every traversed node. Afterwards, upon the second traversal of the node (uphill), it moves to the right child. Finally, at the third traversal of the node (uphill but coming from the right child), the node is visited.



Visiting order:
A, C, E, D, B, H, I, G, F

Tree traversal in depth: *depth-first search* `post-order` — pseudocode

Algorithm: iterative post-order

```
procedure ITERATIVEPOSTORDER(root)
    s ← EmptyStack
    v ← root;   lastV ← null;
    while (true) do
        if (v $\neq$ null) then
            s.push(v);   v ← v.left;
        else
            if (s.IsEmpty) then return; end if
            peekv ← s.peek
            if (peekv.right $\neq$ null and lastV $\neq$ peekv.right) then
                v ← peekv.right
            else
                visit(peekv);   lastV ← s.pop;
            end if
        end if
    end while
end procedure
```

`peek` retrieves the information contained in the top element of the stack (the last entered) but *it does not remove this item from the stack*, as `pop` does.

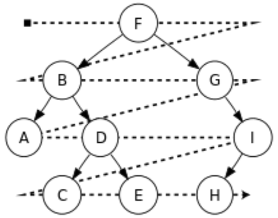
Algorithm: recursive post-order

```
procedure POSTORDER(v)
    if (v = null) then
        return
    end if
    postorder(v.left)
    postorder(v.right)
    visit(v)
end procedure
```

Tree traversal by levels: `breadth-first search`

Description

The nodes are listed by depth level, giving priority to the left.



Visiting order:
F, B, G, A, D, I, C, E, H

Algorithm: breadth-first search (iterative)

```
procedure BREADTHFIRSTSEARCH(root)
    q ← EmptyQueue
    q.enqueue(root)
    while (not q.IsEmpty) do
        node ← q.dequeue
        visit(node)
        if (node.left $\neq$ null) then
            q.enqueue(node.left)
        end if
        if (node.right $\neq$ null) then
            q.enqueue(node.right)
        end if
    end while
end procedure
```

Rooted graphs

Contents

Rooted graphs

A rooted graph (also called a pointed graph or a flow graph) is a graph in which one vertex has been distinguished as the root.

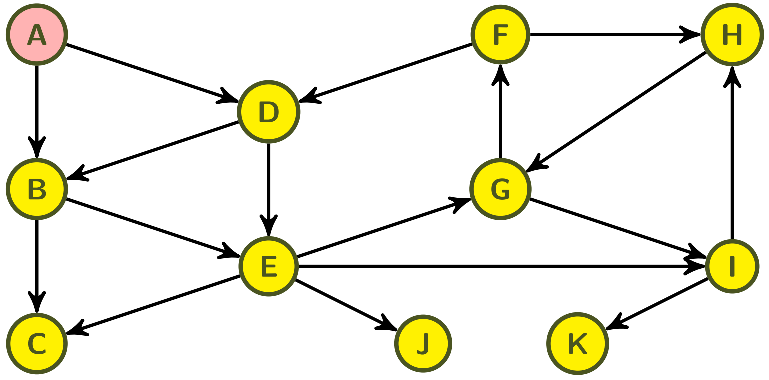
Definitions

A *leaf* of a rooted graph is any vertex of valence 1 different from the root. A *branching vertex* is any vertex with valence greater than two that is different from the root.

Remark: *The valence of a vertex does not depend on the root*

Therefore, *the leaves and branching vertices of a graph are independent of the root node except the root node itself* (which abandons its leaf or branching character when it is designated to be the root).

An example of a rooted graph (with root **A**)



Examples on the definitions

This rooted graph has the vertices *J* and *K* as leaves, and *B*, *D*, *E*, *F*, *G*, *H* and *I* as branching vertices.

Depths in rooted graphs

Definition: Depth of a vertex

In a rooted graph the depth of a vertex is defined as the ***minimum distance*** *from the root to the selected vertex.*

Minimum distance:

Given two vertices α and β, the *minimum distance from α to β* is measured as the shortest length of a path from α to β (note that in a graph there may be more than one of these paths — there may even be more than one path of minimum length from α to β).

Obviously the distance of a node to itself is 0.

On the other hand, in a graph there may be no path from α to β. In this case the distance from α to β is ∞ by convention.

Definition: Depth of a rooted graph

The depth of a rooted graph is defined as the maximum depth of the vertices.

Rooted graphs — Depth properties

From the above definitions and examples it follows:

- The depth of the root is 0.
- The depths of the vertices of a rooted graph depend on the chosen root.
- Therefore, the depth of a rooted graph depends on the root.
- The computation of the depth of a rooted graph (and therefore of the depths of all its vertices) requires the computations of the minimum distances (shorter paths) from the root to each node.
- In a connected undirected rooted graph all nodes have finite depth. However, on a directed rooted graph (even if it is connected) there might exist nodes with infinite depth.

Rooted graphs traversal: Breadth-first search

Contents

1. Rooted graphs traversal and the depth function: breadth-first search algorithm
2. An example
3. Comments on the depth function for rooted graphs
4. Comments on graph traversal with *breadth-first search*
5. Spanning Trees and Minimal Spanning Trees
6. An implementation of the breadth-first search algorithm in **C**

Rooted graphs traversal and the depth function: the *breadth-first search algorithm*

Graph traversal or *graph search* refers to the process of visiting each vertex in a graph. Such traversals are classified by the order in which the vertices are visited.

A *breadth-first search* (BFS) is a technique for traversing a finite graph. BFS visits the sibling vertices before visiting the child vertices, and a queue is used in the search process.

This algorithm finds *a* shortest path from the root of the graph to every one of its vertices, thus computing the depths of all vertices.

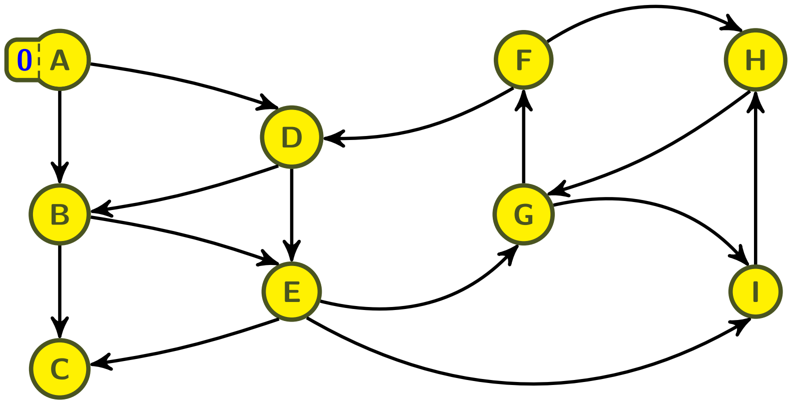
The Breadth-first search algorithm

Pseudocode of Breadth-first search with parents memory for graphs

```
procedure BFS(graph G, order, source, parent[order])
    depth[order] ← initialized to ∞        ▷ To store the depths of all nodes, and to control
                                             whether a node has been already visited
    q ← EmptyQueue                          ▷ Initialization
    q.enqueue(source)                       ▷ Initialization
    depth[source] ← 0                       ▷ source has depth 0 and it has been already enqueued (≠ ∞)
    parent[source] ← ∞                      ▷ source has no parent
    while (not q.IsEmpty) do
        node ← q.dequeue
        for each adj ∈ node.successors do
            if (depth[adj] = ∞) then        ▷ adj has not been enqueued (visited) previously
                q.enqueue ← adj
                depth[adj] ← depth[node]+1  ▷ node is already visited (depth[node] ≠ ∞)
                                              depth[adj] is derived from depth[node]
                parent[adj] ← node          ▷ Setting parent as the node arriving to adj
            end if
        end for
    end while
end procedure
```

An example of the Breadth-first search algorithm

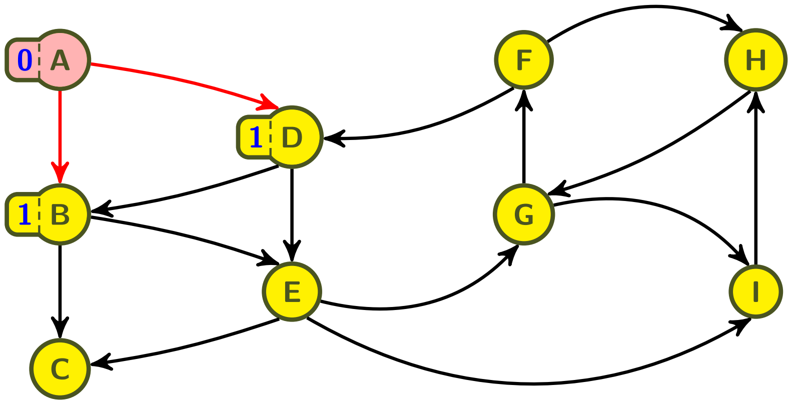
Computing a minimal spanning tree



Q	A
depth	0
parent	nil

An example of the Breadth-first search algorithm

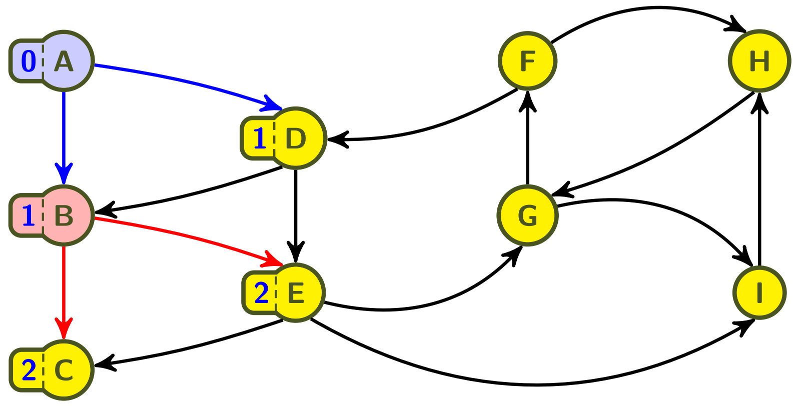
Computing a minimal spanning tree



Q	A	B	D
depth	0	1	1
parent	nil	A	A

An example of the Breadth-first search algorithm

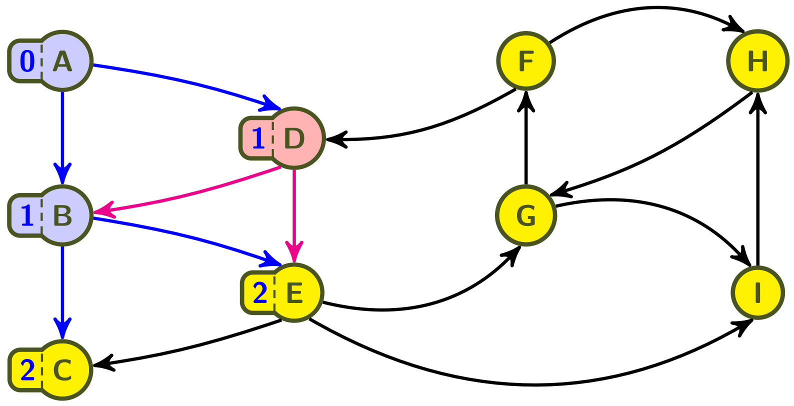
Computing a minimal spanning tree



Q	A	**B**	D	C	E
depth	0	1	1	2	2
parent	nil	A	A	B	B

An example of the Breadth-first search algorithm

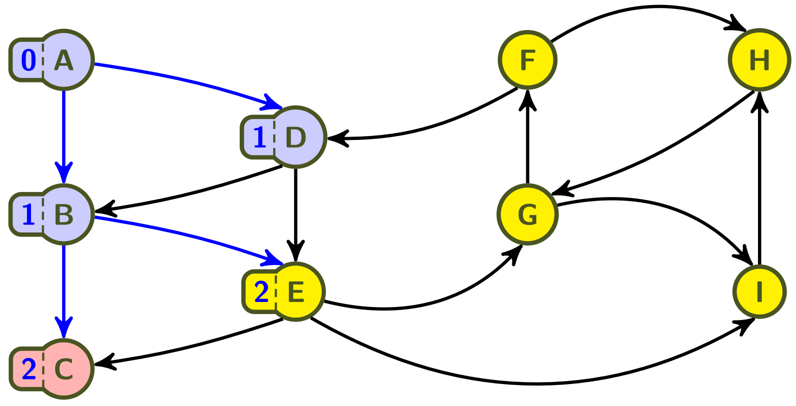
Computing a minimal spanning tree



Q	A	B	**D**	C	E
depth	0	1	1	2	2
parent	nil	A	A	B	B

An example of the Breadth-first search algorithm

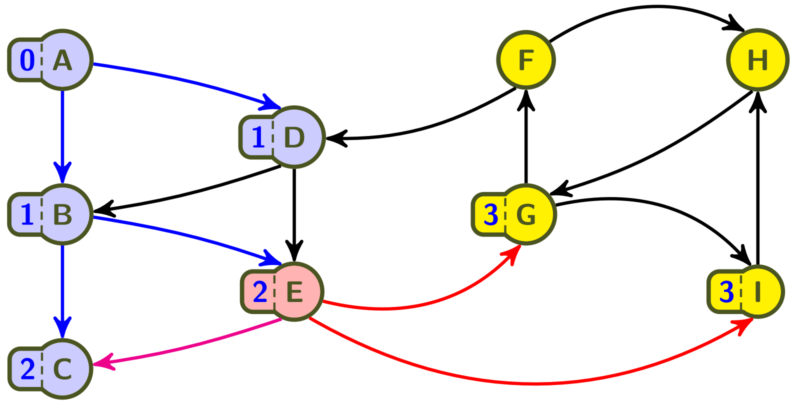
Computing a minimal spanning tree



Q	A	B	D	**C**	E
depth	0	1	1	2	2
parent	nil	A	A	B	B

An example of the Breadth-first search algorithm

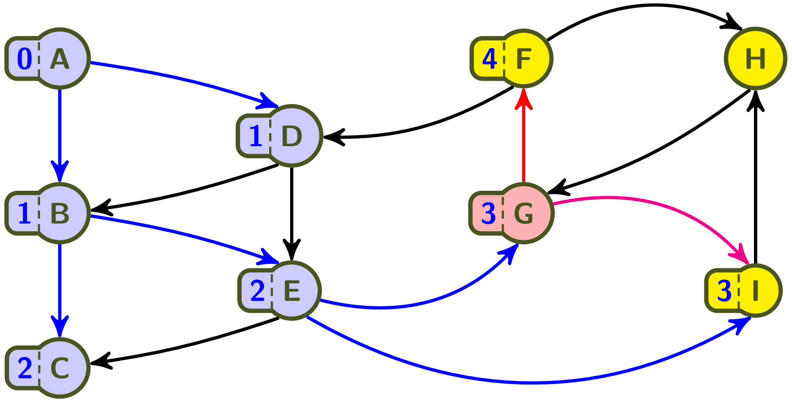
Computing a minimal spanning tree



Q	A	B	D	C	**E**	G	I
depth	0	1	1	2	2	3	3
parent	nil	A	A	B	B	E	E

An example of the Breadth-first search algorithm

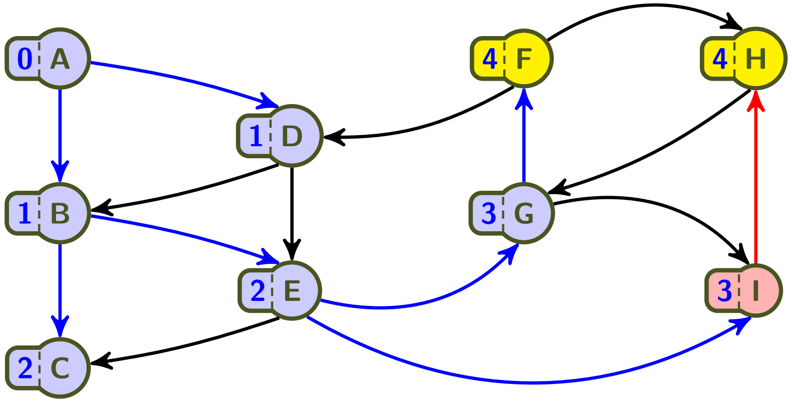
Computing a minimal spanning tree



Q	A	B	D	C	E	**G**	I	F
depth	0	1	1	2	2	3	3	4
parent	nil	A	A	B	B	E	E	G

An example of the Breadth-first search algorithm

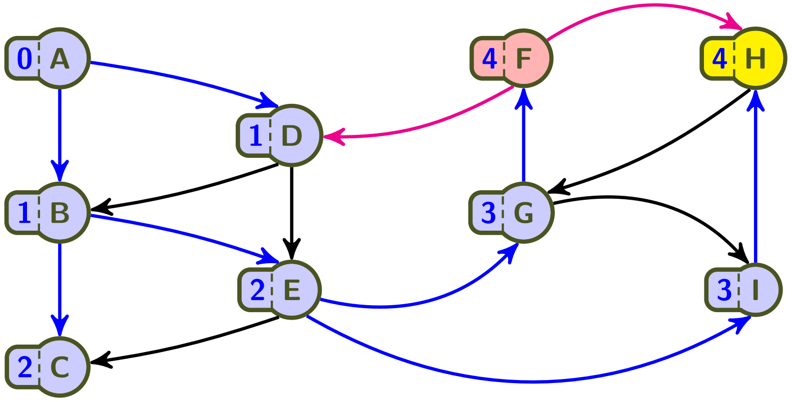
Computing a minimal spanning tree



Q	A	B	D	C	E	G	**I**	F	H
depth	0	1	1	2	2	3	3	4	4
parent	nil	A	A	B	B	E	E	G	I

An example of the Breadth-first search algorithm

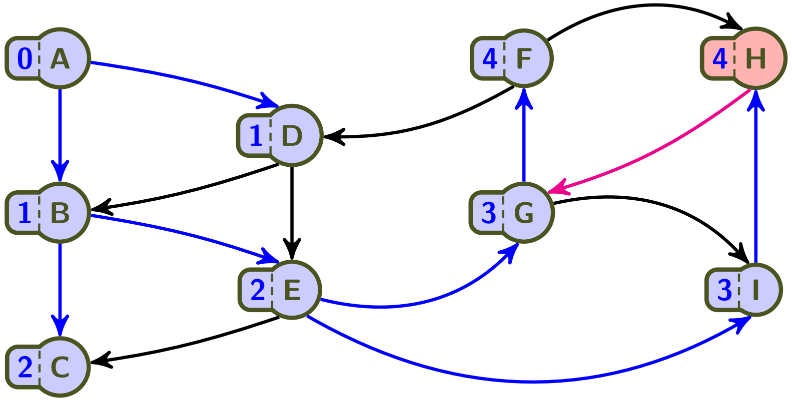
Computing a minimal spanning tree



Q	A	B	D	C	E	G	I	**F**	H
depth	0	1	1	2	2	3	3	4	4
parent	nil	A	A	B	B	E	E	G	I

An example of the Breadth-first search algorithm

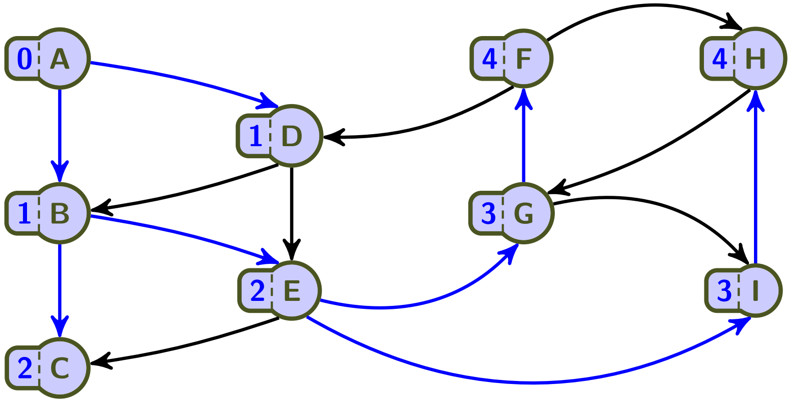
Computing a minimal spanning tree



Q	A	B	D	C	E	G	I	F	**H**
depth	0	1	1	2	2	3	3	4	4
parent	nil	A	A	B	B	E	E	G	I

An example of the Breadth-first search algorithm

Computing a minimal spanning tree



Q	A	B	D	C	E	G	I	F	H
depth	0	1	1	2	2	3	3	4	4
parent	nil	A	A	B	B	E	E	G	I

Comments on the depth function for rooted graphs

All depths

The depth of every node in the rooted graph of the previous example is listed in the table, and included in the graph itself (the number at the left of every node's box).

The depth of the whole graph is **4**.

Example: Why the depth of node G in the above graph is 3

Recall that in a rooted graph the *depth* of a vertex is defined as the minimum distance from the root to the selected vertex. So, let us write some of the paths from A to G in the above graph.

- $A \longrightarrow B \longrightarrow E \longrightarrow G$ ▷ *a shortest path from A to G — of length 3*
- $A \longrightarrow D \longrightarrow E \longrightarrow G$ ▷ *another shortest path from A to G — recall the non-unicity*
- $A \longrightarrow D \longrightarrow B \longrightarrow E \longrightarrow G$ ▷ *wrong: $A \to D \to B$ is not minimal*
- $A \longrightarrow D \longrightarrow B \longrightarrow E \longrightarrow I \longrightarrow H \longrightarrow G$
- $A \longrightarrow D \longrightarrow E \longrightarrow G \longrightarrow I \longrightarrow H \longrightarrow G$ ▷ *non-sense turning around a circuit*
- $A \to D \to E \to G \to I \to H \to G \to I \to H \to G$ ▷ *more non-sense circuit turning*

Comments on graph traversal with *breadth-first search*

Remark

The *breadth-first search* algorithm finds *a* shortest path from the root of the graph to every one of its vertices, thus computing the depths of all vertices.

Remark

This is *NOT* equivalent to the computation of the shortest path between *any pair of nodes* of the graph.

In fact the BFS algorithm returns a *minimal spanning tree*.

Example

The minimal spanning tree of the previous example is the one (whose arrows are) marked in blue.

Spanning Tree

Definition

A *spanning tree* of a connected graph is a subset of edges of the graph that *connects all its vertices*, and is a tree.

Remarks

- A non-connected graph has no spanning tree (since trees are connected and spanning trees must contain all vertices).
- **Non-unicity**: A graph can have more than one spanning tree.
- If all edges of a graph are also edges of a spanning tree, then the graph coincides with its spanning tree. Therefore, any tree coincides with its (unique) spanning tree.
- A spanning tree of a connected graph can also be defined both as a maximal set of edges that do not contain cycles, or as a minimal set of edges connecting all vertices.
- In particular, a connected graph always has a spanning tree. Moreover, this spanning tree can be obtained by deleting edges so that each deletion breaks a circuit (loop).

Minimal Spanning Tree

Definition

A *minimal spanning tree* of a connected graph is a spanning tree such that every path α in the spanning tree, starting at the root vertex, has minimum length among all paths in the graph starting at the root and ending at the same vertex as α.

Example

See the spanning tree (marked in blue) in the previous example.

Implementation of the *breadth-first search* algorithm in C

The main BFS code

Remark: `USHRT_MAX` is the largest number that can be stored in an `unsigned short` variable. In other words, in the "world" of `unsigned short`'s, `USHRT_MAX` is ∞.

```
typedef struct { char name;  unsigned short nsucc, successors[3]; } graph_node;

void graph_node_print(graph_node *Graph, unsigned short v, unsigned short d, unsigned short p){
   char static head_print = 0;
   if(!head_print){ head_print = 1;
       fprintf(stdout, "Visit | Depth |\nOrder | found | Parent\n------|-------|-------\n"); }
   fprintf(stdout, "%c (%u) |%4u   |", Graph[v].name, v, d);
   if(p != USHRT_MAX) fprintf(stdout, "%2c (%u)", Graph[p].name, p);
   fprintf(stdout, "\n");
}

void BFS( graph_node *Graph, unsigned short order, unsigned short source ){
    Queue Q = { NULL, NULL };
    unsigned short depth[order], parent[order];
    memset(depth, USHRT_MAX, order*sizeof(unsigned short));
    enqueue(source, &Q); depth[source]=0U; parent[source] = USHRT_MAX;

    while( !IsEmpty(Q) ){ register unsigned short v, i, s;
        v = dequeue(&Q); graph_node_print(Graph, v, depth[v], parent[v]);
        for(i=0; i < Graph[v].nsucc; i++) {
            s = Graph[v].successors[i];
            if(depth[s] == USHRT_MAX){ enqueue(s, &Q); depth[s] = depth[v] + 1; parent[s] = v; }
}}}

int main (void) {
graph_node GraphDem[9] = { {'A', 2, {1,3}}, {'B', 2, {2,4}}, {'C', 0, {}}, {'D', 2, {1,4}},
     {'E', 3, {2,6,8}}, {'F', 2, {3,7}}, {'G', 2, {5,8}},   {'H', 1, {6}}, {'I', 1, {7}} };

     BFS(GraphDem, 9U, 0U);
}
```

Direct assignment at declaration time as initialization.

Output: depths and the spanning tree

```
Visit | Depth |
Order | found | Parent
------|-------|-------
A (0) |   0   |
B (1) |   1   | A (0)
D (3) |   1   | A (0)
C (2) |   2   | B (1)
E (4) |   2   | B (1)
G (6) |   3   | E (4)
I (8) |   3   | E (4)
F (5) |   4   | G (6)
H (7) |   4   | I (8)
```

Implementation of the *breadth-first search* algorithm in C

Initializations and queue functions

```
#include <stdio.h>
#include <stdlib.h> // For exit() and malloc()
#include <limits.h> // For USHRT_MAX
#include <string.h> // For memset

typedef struct QueueElementstructure {
   unsigned short vertex;
   struct QueueElementstructure *seg;
} QueueElement;

typedef struct { QueueElement *start, *end; } Queue;
int IsEmpty( Queue Q ){ return ( Q.start == NULL ); }

int enqueue( unsigned short vert2Q, Queue *Q ){
  QueueElement *aux = (QueueElement *) malloc(sizeof(QueueElement)); if( aux == NULL ) return 0;

  aux->vertex=vert2Q; aux->seg=NULL;
  if( Q->start ) Q->end->seg=aux; else Q->start=aux;
  Q->end = aux;
  return 1;
}

unsigned int dequeue( Queue *Q ){ if( IsEmpty(*Q) ) return UINT_MAX;
  QueueElement *node_inicial = Q->start;
  unsigned int v = node_inicial->vertex;

  Q->start = Q->start->seg;
  free(node_inicial);
  return v;
}
```

The number of elements `nel` of the queue is not used in this application. Hence, it is omitted.

Similarly, the function to initialise the queue is not necessary as we use direct assignment when declaring the queue:

`Queue Q = { NULL, NULL };`

Implementation of the BFS in C: an exercise

Optional Exercise

By using the parents vector, create a procedure that writes the shortest path found by BFS from the root to each of the nodes.

Remark

- It is more compact (but more difficult) to find first the leaves of the spanning tree, and then write the minimum paths from the root to each of these nodes.
- To compute the paths, go backwards (from children to parents — starting at the end of the paths) and then reverse the paths.

Rooted graphs traversal: Depth-first search

Contents

Rooted graphs traversal: Depth-first search

The `depth-first search` algorithm for graphs

does not work as for trees, and likewise it is not similar to the `breadth-first search` algorithm for graphs (page 52) with queues replaced by stacks.

Remarks: on the `depth-first search` algorithm for graphs

- It traverses successfully the graph "in depth" (see the example starting on page 74) though the order of visit of the nodes depends on the combinatorics of the graph and the order in which adjacent nodes are expanded.
- It does not compute the depth function correctly.
- It correctly computes a spanning tree, although in this case it is not necessarily minimal (since it does not calculate well the depth function).

The Depth-first search algorithm

Pseudocode of Depth-first search with parents memory for graphs

```
procedure DFS(graph G, order, root, parent[order])
    visited[order] ← initialized to false      ▷ To control whether a node has been visited or not.
    s ← EmptyStack
    s.push(root)
    parent[root] ← ∞                           ▷ The root has no parent.
    while (not s.IsEmpty) do
        node ← s.pop
        if (visited[node]) then continue       ▷ A vertex may have been placed several times in
                                                 the stack. When processing the first of these
                                                 instances the vertex is visited. After this, the re-
                                                 maining instances of the vertex must be ignored.
        visited[node] ← true
        visit(node)
        for each adj ∈ node.successors do      ▷ It determines the order in which the nodes are
                                                 visited, their parents, and a spanning tree.
            if (not visited[adj]) then         ▷ Visited nodes do not need to be re-visited.
                parent[adj] ← node
                s.push(adj)
            end if
        end for
    end while
end procedure
```

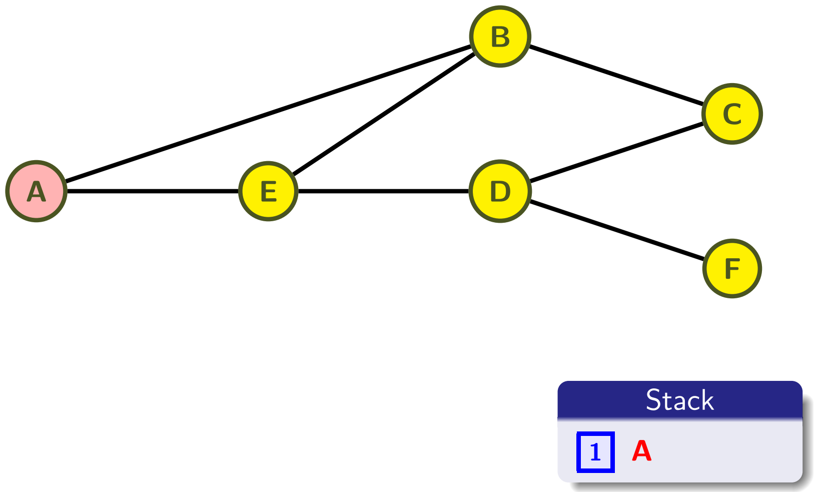
A node v can be placed in the stack several times

since it can be adjacent to several nodes explored but not visited, and each of these nodes adds a copy of v to the stack. Obviously we will only remove one of these copies from the stack: the last one we added.

An example of the Depth-first search algorithm

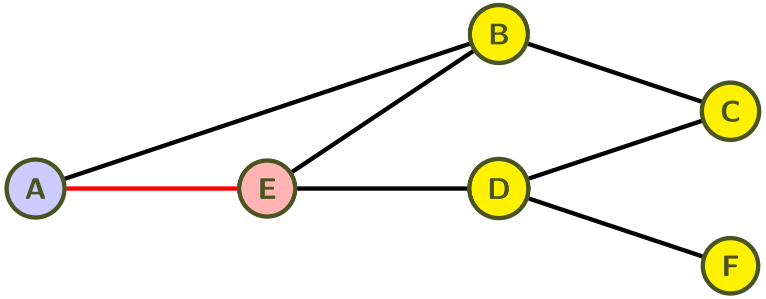
The undirected graph from page 2 (with vertices labeled with capital letters for clarity)

Finding a spanning tree (not necessarily minimal)



An example of the Depth-first search algorithm

The undirected graph from page 2 (with vertices labeled with capital letters for clarity)
Finding a spanning tree (not necessarily minimal)



Visited nodes

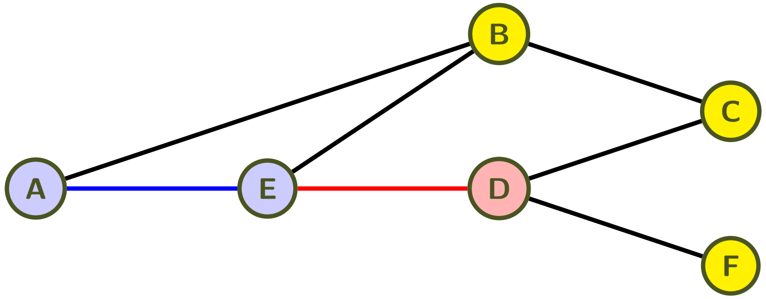
ordering	1
node	**A**
parent	

Stack

$\boxed{2}$ **E B**

An example of the Depth-first search algorithm

The undirected graph from page 2 (with vertices labeled with capital letters for clarity)
Finding a spanning tree (not necessarily minimal)



Visited nodes

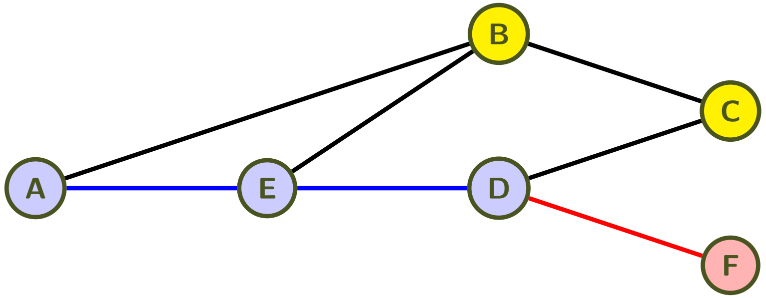
ordering	1	2
node	**A**	**E**
parent		A

Stack

$\boxed{3}$ **D B B**

An example of the Depth-first search algorithm

The undirected graph from page 2 (with vertices labeled with capital letters for clarity)
Finding a spanning tree (not necessarily minimal)



Visited nodes

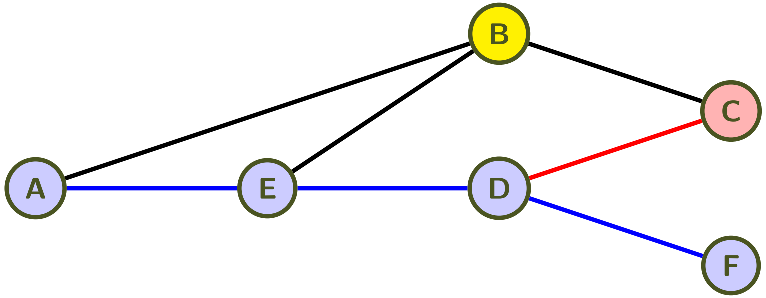
ordering	1	2	3
node	**A**	**E**	**D**
parent		A	E

Stack

$\boxed{4}$ **F C B B**

An example of the Depth-first search algorithm

The undirected graph from page 2 (with vertices labeled with capital letters for clarity)
Finding a spanning tree (not necessarily minimal)



Visited nodes

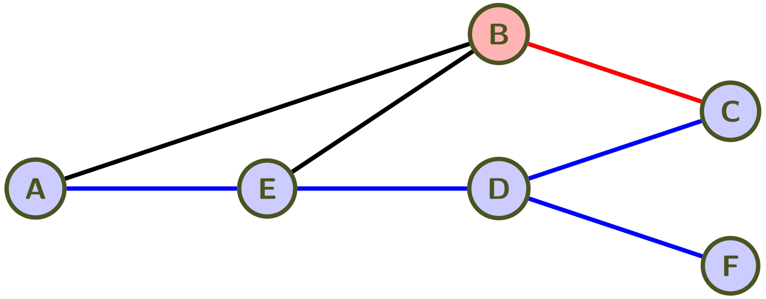
ordering	1	2	3	4
node	**A**	**E**	**D**	**F**
parent		A	E	D

Stack

$\boxed{5}$ **C B B**

An example of the Depth-first search algorithm

The undirected graph from page 2 (with vertices labeled with capital letters for clarity)
Finding a spanning tree (not necessarily minimal)



Visited nodes

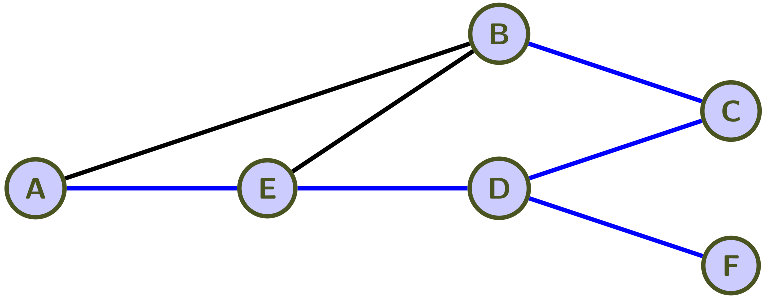
ordering	1	2	3	4	5
node	**A**	**E**	**D**	**F**	**C**
parent		A	E	D	D

Stack

6 B B B

An example of the Depth-first search algorithm

The undirected graph from page 2 (with vertices labeled with capital letters for clarity)
Finding a spanning tree (not necessarily minimal)



Visited nodes

ordering	1	2	3	4	5	6
node	**A**	**E**	**D**	**F**	**C**	**B**
parent		A	E	D	D	C

Stack

7 B B

Comments on graph traversal with *depth-first search*

The *depth-first search* algorithm

traverses the whole graph "in depth" visiting all graph's nodes but it does not compute the depth function correctly.

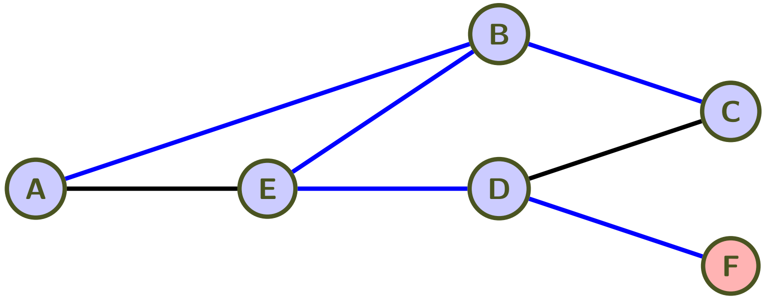
In fact the DFS algorithm returns a spanning tree (which is not necessarily minimal).

Example

The spanning tree of the previous example is the one (whose arrows are) marked in blue.

A second example of the Depth-first search algorithm

The same graph as before (the undirected graph from page 2) with a diffrent root



Visited nodes

ordering	1	2	3	4	5	6
node	**F**	**D**	**E**	**B**	**C**	**A**
parent		F	D	E	B	B

Implementation of the *depth-first search* algorithm in **C**

The main DFS code

Remark: `USHRT_MAX` is the largest number that can be stored in an `unsigned short` variable. In other words, in the "world" of `unsigned short`'s, `USHRT_MAX` is ∞.

```
typedef unsigned short u_short;
typedef struct { char name; u_short nsucc, successors[3]; } graph_node;
void graph_node_print(graph_node *G, u_short v, u_short p, u_short s){
   if(v == s) { fprintf(stdout,
        "\n\n ordering | parent\n----------|-------\n%4c (%u)  |\n", G[v].name, v
   ); } else  fprintf(stdout, "%4c (%u)  | %c (%u)\n", G[v].name, v, G[p].name, p);
}

void DFS( graph_node *Graph, u_short order, u_short source ){
    register u_short i;
    u_short parent[order];
    Stack St = NULL;
    char visited[order]; memset(visited, 0, order);
    push(source, &St); parent[source] = USHRT_MAX;

    while(!IsEmpty(St)){ u_short node = pop(&St);
      if(visited[node]) continue;
      visited[node] = 1;
      graph_node_print(Graph, node, parent[node], source);
      for(i=0; i < Graph[node].nsucc; i++) {
          u_short adj = Graph[node].successors[i];
          if(!visited[adj]) { push(adj, &St); parent[adj] = node; }
}}}

int main (void) {
    graph_node GrafNO[6] = { {'A', 2, {1, 4}}, {'B', 3, {0, 2, 4}}, {'C', 2, {1, 3}},
                             {'D', 3, {2, 4, 5}}, {'E', 3, {0, 1, 3}}, {'F', 1, {3}} };
     DFS(GrafNO, 6U, 0U);
     DFS(GrafNO, 6U, 5U);
}
```

Output:Traversal and the spanning tree

```
 ordering | parent
----------|-------
   A (0)  |
   E (4)  | A (0)
   D (3)  | E (4)
   F (5)  | D (3)
   C (2)  | D (3)
   B (1)  | C (2)

 ordering | parent
----------|-------
   F (5)  |
   D (3)  | F (5)
   E (4)  | D (3)
   B (1)  | E (4)
   C (2)  | B (1)
   A (0)  | B (1)
```

Implementation of the *depth-first search* algorithm in **C**

Initializations and queue functions

```
#include <stdio.h>
#include <stdlib.h> // For exit() and malloc()
#include <limits.h> // For USHRT_MAX
#include <string.h> // For memset

typedef struct StackElementstructure {
   unsigned short vertex;
   struct StackElementstructure *lower;
} StackElement;

typedef StackElement * Stack;
int IsEmpty( Stack S ){ return ( S == NULL ); }
unsigned short pop( Stack *S ){
    Stack aux = *S;
    unsigned short v = (*S)->vertex;

    *S = (*S)->lower;
    free(aux);

    return v;
}

int push( unsigned short vert2S, Stack *S ){
  StackElement *aux = (StackElement *) malloc(sizeof(StackElement)); if( aux == NULL ) return 0;

  aux->vertex = vert2S;
  aux->lower = *S;
  *S = aux;
  return 1;
}
```

Algorithms for checking the graph connection, and counting the number of connected components

Contents

Undirected graphs: how to detect the connectedness and count the number of connected components

Algorithm: checking the connectedness on undirected graphs

1. Select a random vertex s that will be used as root.
2. Traverse the graph using **BFS** or **DFS**, taking the vertex s as root.
3. At the end of the traversal, check whether we have visited all the vertices of the graph.

The graph is connected if and only if the traversal visits all of its vertices. **The graph**, when connected, **has a unique connected component**.

Algorithm: counting connected components in undirected graphs

The **number of connected components of an undirected graph** coincides with the number of times that the previous algorithm must be iterated (i.e. the *algorithm that checks the connection in undirected graphs*), taking as a root a vertex not visited in the previous iterations, *until we have visited all vertices.*

Note: As we said before, if the first time we apply the *algorithm that checks the connection in undirected graphs* we visit all the vertices of the graph, the graph is connected and therefore it has exactly one connected component.

Directed graphs: weak connection

Obvious observation

To detect if a directed graph is weakly connected, and count its number of weak connected components, the algorithms from the previous page must be used after converting the graph from directed to **non**-directed.

Exercise

Justify how a directed graph can be converted to undirected for each one of the four memory models of graph representation (explained in page 17 and subsequent pages).

Directed graphs: strong connection

Algorithm: inefficiently checking strong connection in directed graphs

Iterate the following for each vertex s in the graph

Procedure

Scroll the graph using **BFS** or **DFS** taking s as root, and check if all vertices of the graph have been visited.
In this case we know that there is a (oriented) path that goes from s to every vertex of the graph.

Otherwise the graph cannot be strongly connected as it exists inaccessible vertices from s.

In summary: a graph is strongly connected if the **procedure** above can be performed for each vertex of the graph, visiting all other vertices at every repetition of the procedure.
Alternatively, the first time that the previous **procedure** fails (i.e. inaccessible vertices are found), we know that the graph is not strongly connected.

Directed graphs: strong connection

An efficient check of the strong connection condition for directed graphs is based on the following

Observation

A directed graph is strongly connected if and only if the following two conditions are verified by every vertex s:

1. There is a (directed) path from s to each vertex of the graph.
2. For each vertex u of the graph there is a directed path from u to s.

Note that condition (2) is equivalent to (1), if we reverse all edges of the graph.

Exercise

Justify how the edges of a directed graph can be reversed, for each one of the four memory models of graph representation (explained in page 17 and subsequent pages).

Directed graphs: strong connection

The above observation gives rise to the following

Algorithm: checking strong connection on directed graphs

1. Select a random vertex s, that we will use as root.
2. Perform a graph traversal by using **BFS** or **DFS**, taking the vertex s as root.
3. At the end of the traversal, check whether we have visited all the vertices of the graph.

   In the negative, the graph is not strongly connected.

   In the affirmative:
   - a. Reverse all edges of the graph.
   - b. Perform a new graph traversal by using **BFS** or **DFS**, taking the vertex s as root.
   - c. At the end of the traversal, check whether we have visited all the vertices of the graph.

   The graph is strongly connected if and only if all the vertices of the graph are visited with the reversed edges.

The graph, when strongly connected, **has exactly one strongly connected component**.

Directed graphs: strong connection

Algorithm:
counting the number of strongly connected components in directed graphs

The **number of strongly connected components of an undirected graph** is the number of times the previous algorithm must be iterated (that is, the *algorithm that checks the strong connection in directed graphs*), taking as a root a vertex not visited in the previous iterations, *until we have visited all vertices*.

Note: If the first time we apply the *algorithm that checks the strong connection in undirected graphs* we visit all the vertices of the graph, the graph is connected and therefore it has a single strongly connected component.