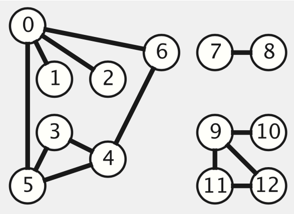
Learning from Networks

Basic Definitions and Algorithms

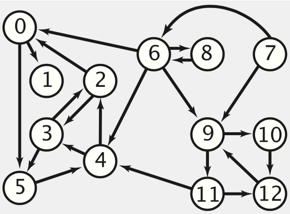
Fabio Vandin

October 2^{nd}, 2024

Graph: Definition



Undirected graph



Directed graph

Definition

Graph $G = (V, E)$

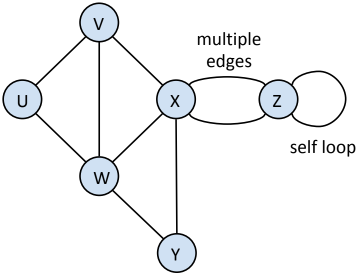
$V \doteq$ set of vertices (sometimes called *nodes*)

$E \doteq$ collection of edges (edge = pair of vertices).

A graph is:

- *directed* if every edge $(u, v) \in E$ is an ordered pair ($u \to v$), sometimes called *arc*
- *undirected* if every edge $(u, v) \in E$ is an unordered pair ($u - v$)

Note



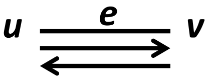
- E is a *collection* and not a *set* since there could be multiple edges u, v
- *self loop:* edge (u, u)
- *simple* graph: without multiple edges and self loops
- *weighted* graph: edges and/or vertices have *weights*

We will mostly focus on simple, unweighted graphs, but will also see some parts on weighted graphs.

Additional Terminology

Let $e = (u, v) \in E$ an edge of a graph G, then:

- e is *incident* to u and to v
- u and v are *adjacent*

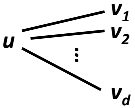


u*, *v **ADJACENT**

e **INCIDENT to *u*/*v***

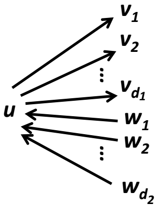
Degree

- undirected graph



d ≐ **DEGREE** of ***u, degree(u)***
num. adjacent vertices = num. incident edges

- directed graph

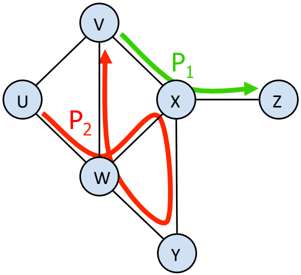


d_1 ≐ **OUTDEGREE** of ***u***

d_2 ≐ **INDEGREE** of ***u***

$d = d_1 + d_2$ ≐ **DEGREE** of ***u***

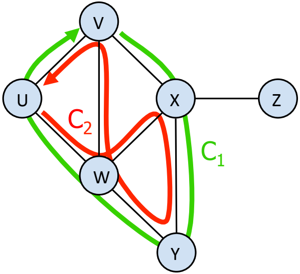
Paths



Path: sequence $u_1, u_2, \ldots, u_k$ of vertices with $(u_i, u_{i+1}) \in E$ for all $1 \leq i < k$.
Example: u, w, x, y, w, v and v, x, z are paths

Simple path: sequence $u_1, u_2, \ldots, u_k$ of vertices *all distinct* with $(u_i, u_{i+1}) \in E$ for all $1 \leq i < k$.
Example: v, x, z is a simple path

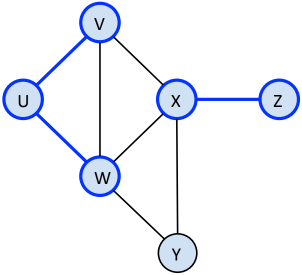
Cycles



Cycle: sequence $u_1, u_2, \ldots, u_k = u_1$ of vertices with $(u_i, u_{i+1}) \in E$ for all $1 \leq i < k$.
Example: u, w, x, y, w, v, u e v, x, y, w, u, v are cycles

Simple cycle: sequence $u_1, u_2, \ldots, u_k = u_1$ of vertices *all distinct* with $(u_i, u_{i+1}) \in E$ for all $1 \leq i < k$.
Example: v, x, y, w, u, v is a simple cycle

Subgraph

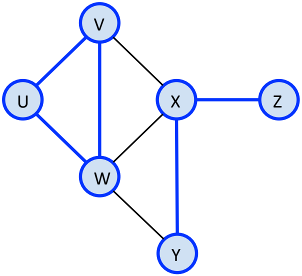


$G' = (V', E')$ is a **subgraph** of $G = (V, E)$ if:

- $V' \subseteq V$
- $E' \subseteq E$
- $\forall (u, v) \in E' : u, v \in V'$

Example: the blue edges and vertices are a subgraph of the (blue/black) graph above

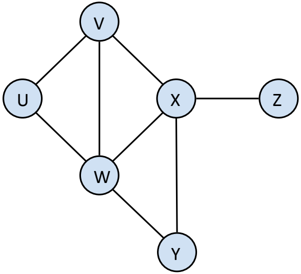
Spanning subgraph



Spanning subgraph: subgraph $G' = (V', E')$ of $G = (V, E)$ such that $V' = V$.

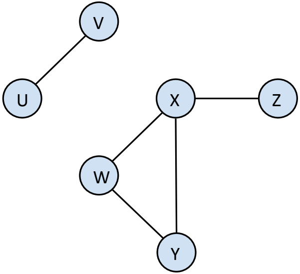
Example: the blue edges and vertices are a spanning subgraph of the (blue/black) graph above

Connected graph



$G = (V, E)$ is *connected* if $\forall u, v \in V$ there exists a path in G starting in u and ending in v

Disconnected graph



$G = (V, E)$ is *disconnected* if there exist $u, v \in V$ such that there is no path in G starting in u and ending in v

Connected components

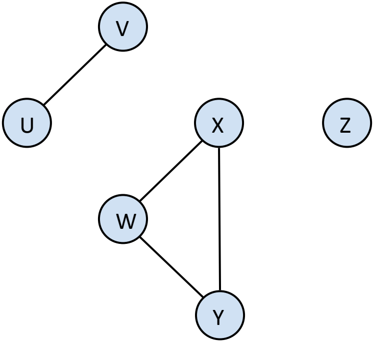
The **connected components** of $G = (V, E)$ are a *partition* of G in subgraphs $\{G_1, G_2, \ldots, G_k\}$, with $G_i = (V_i, E_i)$ such that

- $G_i = (V_i, E_i)$ is connected for all $1 \leq i \leq k$
- $V = V_1 \cup V_2 \cup \cdots \cup V_k$ (*partition* of V)
- $E = E_1 \cup E_2 \cup \cdots \cup E_k$ (*partition* of E)
- $\forall i \neq j$ there are no edges in E between V_i and V_j

Notes:

- $\{G_i : 1 \leq i \leq k\}$ are *maximal* connected subgraphs
- if G is connected $\Rightarrow k = 1$ in the definition above
- the partition of G in connected components is *unique* (up to the ordering of the components)

Connected components: example



There are 3 connected components $G_1 = (V_1, E_1)$, $G_2 = (V_2, E_2)$, $G_3 = (V_3, E_3)$ with

- $V_1 = \{u, v\}$ ($|E_1| = 1$)
- $V_2 = \{x, y, w\}$ ($|E_2| = 3$)
- $V_3 = \{z\}$ ($E_3 = \emptyset$)

Trees

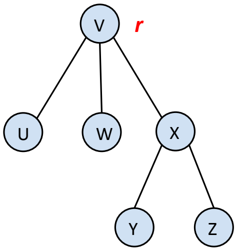
A (free) **tree**) is a graph $G = (V, E)$ *connected* and *without (simple) cycles*

A **rooted tree** is a graph $G = (V, E)$ such that:

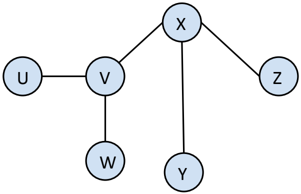
- there exist a *root* vertex $r \in V$
- $\forall u \in V$, with $u \neq r$, there exist a unique vertex $p(u) \in V$ *parent* of u
- $E = \{(u, p(u)) : u \in V, u \neq r\}$
- $\forall u \in V$ walking from parent to parent we reach r

Rooted trees vs Free trees

Rooted tree

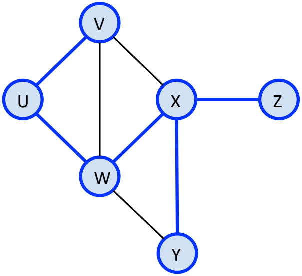


Free tree



The concepts of *rooted tree* and of *free tree* are the *equivalent*: a rooted tree is a free tree, and vice-versa (by choosing any node as the root).

Spanning tree



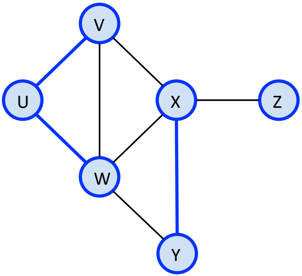
Spanning tree of G: *connected* spanning subgraph of G *without cycles*

Question: does a spanning tree exist for every G?

No: only if G is connected

Example Edges and vertices in blue are a spanning tree of the graph above

Spanning Forest



Spanning forest of G: spanning subgraph without cycles.

Example Edges and vertices in blue are a spanning forest of the graph above

Graph Properties

Let $G = (V, E)$ be a simple and undirected graph with $|V| = n$ and $|E| = m$. Then the following properties hold.

Property

$\sum_{v \in V} degree(v) = 2m$.

Proof

Every edge is counted twice in the sum. □

Property

$m \leq \binom{n}{2}$ ($\Rightarrow m \in O\left(n^2\right)$)

Proof

Since G is simple: E is a subset of the $\binom{n}{2}$ possible pairs of vertices. □

Graph Properties (continue)

Property

If G is a tree then $m = n - 1$

Proof

Consider G as a rooted tree. Then E contains the relations parent-child, that are $n - 1$ (every vertex but the root has a parent). $\square$

Graph Properties (continue)

Property

If G is connected then $m \geq n - 1$

Proof

Consider the following loop:

While($\exists$ cycle C) **do** remove an edge of C from G

Then:

- at the end of every iteration: G is connected
- at the end of the loop: G is connected and without cycles, that is G is a tree with $m' = n - 1 \leq m$ edges.

$\square$

Graph Properties (continue)

Property

If G is a forest (i.e., is without cycles) then $m \leq n - 1$

Proof

Consider the following loop:

While(G is not connected) **do**
add an edge between two connected components G_1, G_2 of G

Then:

- every added edge does not create a cycle in G;
- at the end of the loop: G is connected and without cycles, that is G is a tree with $m' = n - 1 \geq m$ archi.

□

Basic Primitives

- Graph traversal (exploration)
- Graph connectivity
- Finding connected components
- Shortest paths
- Finding a *minimum* spanning tree

Graph Representations: Basic Data Structures

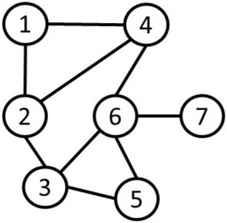
Let $G = (V, E)$ be a graph with $|V| = n$ and $|E| = m$.
Note: sometimes we will assume for simplicity that $V = \{1, 2, \dots, n\}$.

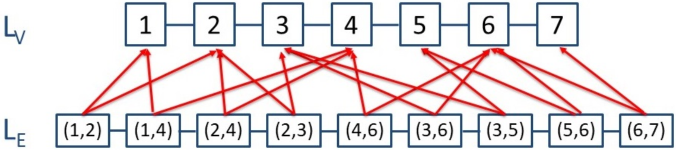
Simplest representation of G uses the following *basic data structures*:

- Vertex list L_V: *every node of the list contains all relevant information for a distinct $v \in V$. If $V = \{1, 2, \dots, n\}$, L_V could be an array.*
- Edge list L_E: *every node of the list contains all the relevant information for a distinct $e = (u, v) \in E$, including "pointers" to u and v.*

Note: Depending on the problem/task, every vertex/edge can be enriched with additional information (ID, weight, labels, ecc.).

Example





Adjacency List Representation

The edge list representation L_E is not useful to develop efficient algorithms. We will now see better and commonly used representations.

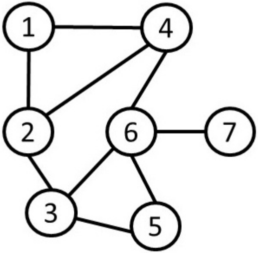
Adjacency List: *For every vertex $v \in V$ there is a list $l(v)$ of pointers to the edges (i.e., elements of L_E) incident to v.*

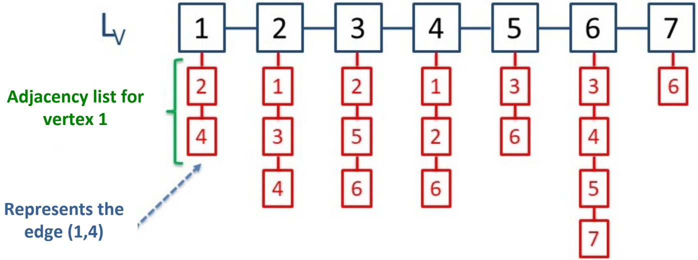
Notes: the representation with adjacency lists is the commonly used for the following reasons:

- it allows to represent G with space linear in the size of G: $\Theta(n + m)$.
- it allows sequential access to the neighbours of a vertex v, in time linear in the degree of v (e.g., iterating on v neighbours...).

Note: we will usually assume the adjacency list representation of a graph G, and to also have L_E.

Example





L_V
1
2
3
4
5
6
7
2
4
1
3
4
2
5
6
1
2
6
3
6
3
4
5
7
6
Adjacency list for vertex 1
Represents the edge (1,4)

Adjacency Matrix Representation

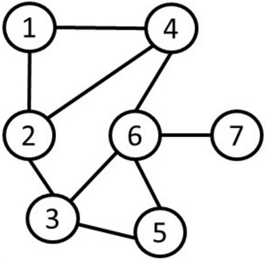
Adjacency matrix A: $n \times n$ matrix such that rows and colums are in 1 to 1 correspondence with G vertices, with

$$A[i_1, i_2] \doteq \begin{cases} \mathit{null} & \text{if } (i_1, i_2) \notin E \\ \text{pointer to } e = (i_1, i_2) \in L_E & \text{if such edge exists} \end{cases}$$

Notes:

- Such structure requires the vertices to be represented by integers (e.g., $V = \{1, 2, \ldots, n\}$) and allows $O(1)$ (constant time) access to an edge and its information.
- the adjacency matrix requires space $\Theta(n^2)$, which can be superlinear in the size of G. Therefore, the representation is used mostly for *dense graphs* (number of edges roughly $\Theta(n^2)$).

Example



They represent the same edge (since the graph is undirected)

	1	2	3	4	5	6	7
1		(1,2)		(1,4)			
2	(2,1)		(2,3)	(2,4)			
3		(3,2)			(3,5)	(3,6)	
4	(4,1)	(4,2)				(4,6)	
5			(5,3)			(5,6)	
6			(6,3)	(6,4)	(6,5)		(6,7)
7						(7,6)	

Graph Traversal of $G = (V, E)$

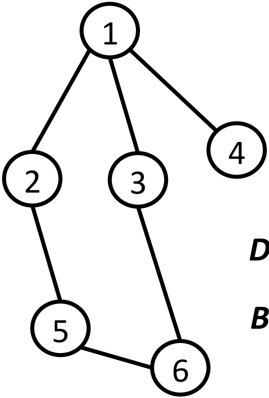
Systematic exploration of G, starting from a vertex s and visiting all vertices.

- Breadth-First Search (BFS): after visiting a vertex, visit all its neighbours before moving to the neighbours of its neighbours
- Depth-First Search (DFS): after visiting a node, visit one of its neighbours, then a neighbour of the neighbour, etc.

Note:

- Why not simply use a scan of the lists L_V e L_E?
- BFS and DFS are *design patterns*: by appropriately choosing what the "visit" of a node is, we can solve several problems: connectivity, finding a spanning tree, etc.

Example



DFS: 1 $\longrightarrow$ 2 $\longrightarrow$ 5 $\longrightarrow$ 6 $\longrightarrow$ 3 $\longrightarrow$ 4

BFS: 1 $\longrightarrow$ 2 $\longrightarrow$ 3 $\longrightarrow$ 4 $\longrightarrow$ 5 $\longrightarrow$ 6

Breadth-First Search: Algorithm BFS

- Iterative algorithm that, starting from a vertex s, "visits" all the vertices in the connected component C_s containing s, "touching" all the vertices and the edges of C_s and partitioning the vertices in levels L_i based on their distance i from s.
- Every vertex $v \in V$ has a field $v.\mathtt{ID}$ with $v.\mathtt{ID} = 1$ if v has been visited, and $v.\mathtt{ID} = 0$ otherwise.
- Every edge $e \in E$ has a field $e.\mathtt{label}$ with $e.\mathtt{label} = \mathtt{null}$ if e has not been labeled yet, or it has a label between `DISCOVERY EDGE` or `CROSS EDGE`

Breadth-First Search: Algorithm BFS

Algorithm `BFS`(G, s)

Input: undirected graph $G = (V, E)$, vertex $s \in V$ not visited
Output: all the vertices of C_s are visited and all the edges C_s labeled as `DISCOVERY` or `CROSS EDGE`

Breadth-First Search: Algorithm BFS

```
visit s; s.ID ← 1;
create list L_0 containing only s; i ← 0;
while (!L_i.isEmpty()) do
    create empty L_{i+1};
    forall v ∈ L_i do
        forall e ∈ G.incidentEdges(v) do
            if (e.label = null) then
                w ← G.opposite(v,e);
                if (w.ID = 0) then
                    e.label ← DISCOVERY EDGE;
                    visit w;
                    w.ID ← 1;
                    insert w in L_{i+1};
                else e.label ← CROSS EDGE;
    i ← i + 1;
return;
```

Example

Analysis of BFS

- Undirected graph $G = (V, E)$, $s \in V$
- $C_s \subseteq G$: connected component containing s.

Proposition

Consider the execution of `BFS(G,s)`. Assume that at the beginning, no vertex of C_s is visited and no edge of C_s is labeled. At the end of the execution:

1. all vertices of C_s are visited and all edges of C_s are labeled as `DISCOVERY` or `CROSS EDGE`;
2. the `DISCOVERY EDGES` are a spanning tree T of C_s rooted in s, called *BFS tree*;
3. $\forall v \in L_i$ the path in T from s to v has i edges and i is the minimum number of edges among any path from s to v in G; i is the *distance* between s and v;
4. if $(u, v) \in E$ and $(u, v) \notin T$ ($\Rightarrow (u, v)$ is a CROSS EDGE) then the indices of u e v differ by at most 1.

Proof: see the material online.

Complexity of BFS(G, s)

Theorem

Consider the execution of `BFS(`G`,`s`)`. Assume that at the beginning, no vertex of C_s is visited and no edge of C_s is labeled. The complexity of `BFS(`G`,`s`)` is $\Theta(m_s)$, where m_s is the number of edges of C_s.

Corollary

If $G = (V, E)$ is connected, `BFS(`G`,`s`)` has complexity $\Theta(|E|) \;\; \forall s \in V$.

Proof of the Theorem

Visiting the Whole Graph

Note that BFS(G, s) visits only the connected component C_s of s.

How can we visit the whole graph?

This is a common *design pattern* to solve problems using the BFS.

forall $v \in V$ **do** $v.\mathtt{ID} \leftarrow 0$;
forall $v \in V$ **do**
    if $(v.\mathtt{ID} = 0)$ **then**
        BFS(G, v);

Note: We assume that there is an iterator on L_V that allows to access the next vertex in time $O(1)$.

Visiting the Whole Graph (Analysis)

Applications of BFS: Connectivity

Input: Graph $G = (V, E)$.
Output: Number of connected components of G and each connected components with a different ID values for its vertices.

Note.: *If the return value is 1, then G is connected.*

Let BFS(G,v,k) be a modified version of BFS(G, v), where the instruction

$$w.\mathtt{ID} \leftarrow 1$$

is substituted by

$$w.\mathtt{ID} \leftarrow k$$

The following algorithm computes the number of connected components of G and assigns to all vertices of each connected component the same value ID.

for $v \leftarrow 1$ *to* n **do** $v.\mathtt{ID} \leftarrow 0$;
$k \leftarrow 0$;
for $v \leftarrow 1$ *to* n **do**
    if ($v.\mathtt{ID} = 0$) **then**
        $k \leftarrow k + 1$;
        BFS(G,v,k);
return k

Applications of BFS: Connectivity (Analysis)

The following proposition can be easily proved.

Proposition

Given $G = (V, E)$, with $|V| = n$ and $|E| = m$, the following problems can be solved in time $O(m + n)$ using the BFS:

1. test if G is connected;
2. find the connected components of G;
3. find a spanning tree of G, if G is connected:
4. find a shortest path between vertices s and t, if it exists;
5. find a cycle, if it exists.

Depth-First Search: Algorithm

- Recursive algorithm: starting from vertex s, visits all the vertices in the connected component of C_s of s and labeling all edges of C_s
- Every vertex $v \in V$ has a field $v.\mathtt{ID}$, with $v.\mathtt{ID} = 1$ if v has been visited, and $v.\mathtt{ID} = 0$ otherwise.
- Every edges $e \in E$ has a filed $e.\mathtt{label}$ with $e.\mathtt{label} = \mathtt{null}$ if e has not been labeled, and one between the labels `DISCOVERY EDGE` or `BACK EDGE` otherwise.
- During the execution of the algorithm, we say that a vertex u is discoverable from v if there exists a path from u to v made of vertices not already visited.

Depth-First Search: Algorithm

Algorithm DFS(G, v) (*first call:* $v = s$)

Input: undirected graph $G = (V, E)$, vertex $v \in V$ not visited
Output: all vertices *discoverable* from v visited, and all edges incident to them labeled as DISCOVERY or BACK EDGE

```
visit v; v.ID ← 1;
forall e ∈ G.incidentEdges(v) do
    if (e.label = null) then
        w ← G.opposite(v,e);
        if (w.ID = 0) then
            e.label ← DISCOVERY EDGE;
            DFS(G,w);
        else e.label ← BACK EDGE;
return;
```

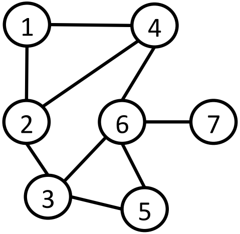
Assumptions for Implementation

- `incidentEdges(`v`)` returns an iterators on the neighbours of v (i.e., on the elements of the adjacency list of v), and the **forall** loop access them iteratively, in time $\Theta(1)$ for each of them
- `opposite(`v`,`e`)` returns the vertex of e opposite to v, in time $\Theta(1)$
- when the first call `DFS(`G`,`s`)` is made, all vertices are not visited (v`.ID`$= 0$ for all $v \in V$), and all edges are not labeled (e`.label=null` for all $e \in E$)

Note: for a generic call `DFS(`G`,`v`)` some vertices of the connected component of v may be already visited and some edges may be already labeled.

Example

Example



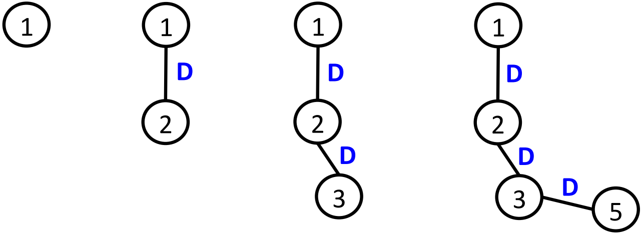
s = 1

Adjacency lists

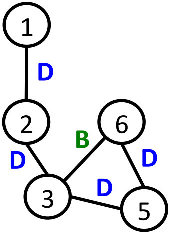
1:	2	4		
2:	1	3	4	
3:	2	5	6	
4:	1	2	6	
5:	3	6		
6:	3	4	5	7
7:	6			

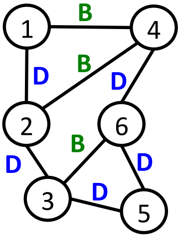
D = DISCOVERY EDGE

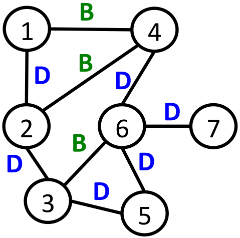
B = BACK EDGE



Example (continue)







D = DISCOVERY EDGE

B = BACK EDGE

Analysis of $\texttt{DFS}(G,s)$

- $G = (V, E)$ undirected graph, $s \in V$
- $C_s \subseteq G$: connected component of G containing s.

Proposition

Assume to execute $\texttt{DFS}(G,s)$ and that, at the beginning, the vertices and edges of C_s are not visited and not labeled. At the end of the execution:

1. all the vertices of C_s are visited and all the edges of C_s are labeled as `DISCOVERY` o `BACK EDGE`;
2. the `DISCOVERY EDGES` are a spanning tree T of C_s rooted in s.

Proof: see the material online.

Complexity of DFS(G, s)

Theorem

Assume to execute DFS(G,s) and that, at the beginning, the vertices and edges of C_s are not visited and not labeled. The complexity of DFS(G,s) is $\Theta(m_s)$, where m_s is the number of edges of C_s.

Note: if $G = (V, E)$ is connected, the complexity of DFS(G,s) is $\Theta(|E|)$.

Similarly to what we have seen for the BFS, the following result is easy to prove.

Proposition

Given $G = (V, E)$, with $|V| = n$ and $|E| = m$, the following problems can be solved in time $O(m + n)$ using the DFS:

1. test if G is connected;
2. find the connected components of G;
3. find a spanning tree of G, if G is connected:
4. find a path between s and t, if it exists (s-t reachability);
5. find a cycle, if it exists.