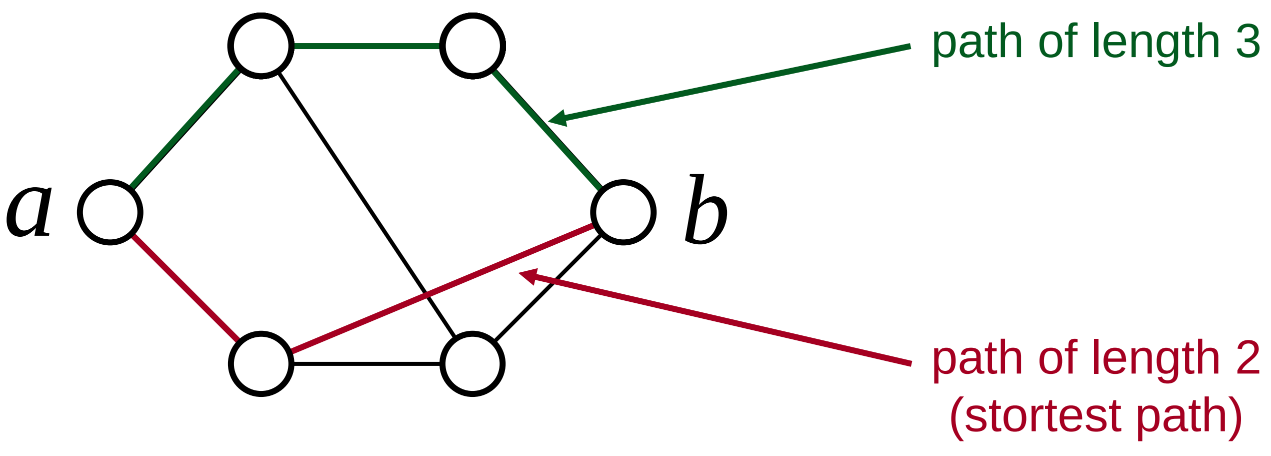
Shortest paths

- The length of a path connecting two vertices a, b :
  - the number of the edges in the path
- The distance of two vertices a, b in a graph:
  - the smallest length of a path that connects a and b
  - e.g.: two adjacent vertices have distance 1
- The shortest path problem:
  - given two vertices a and b, what is their distance?

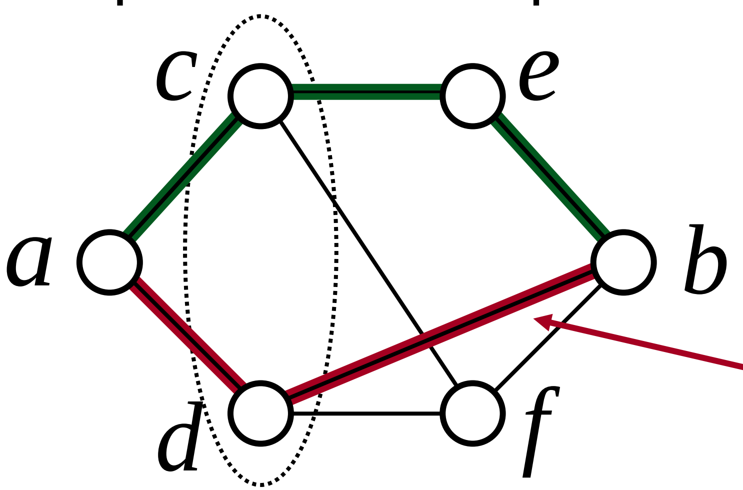


Shortest paths

Sketch of a simple algorithm for shortest paths:

- you stand on a vertex a of the graph and need to find your distance to vertex b
- ask all your neighbours what is their distance to b and compute the smallest of these distances, say x
- then your distance to b is equal to $x+1$

The previous example:



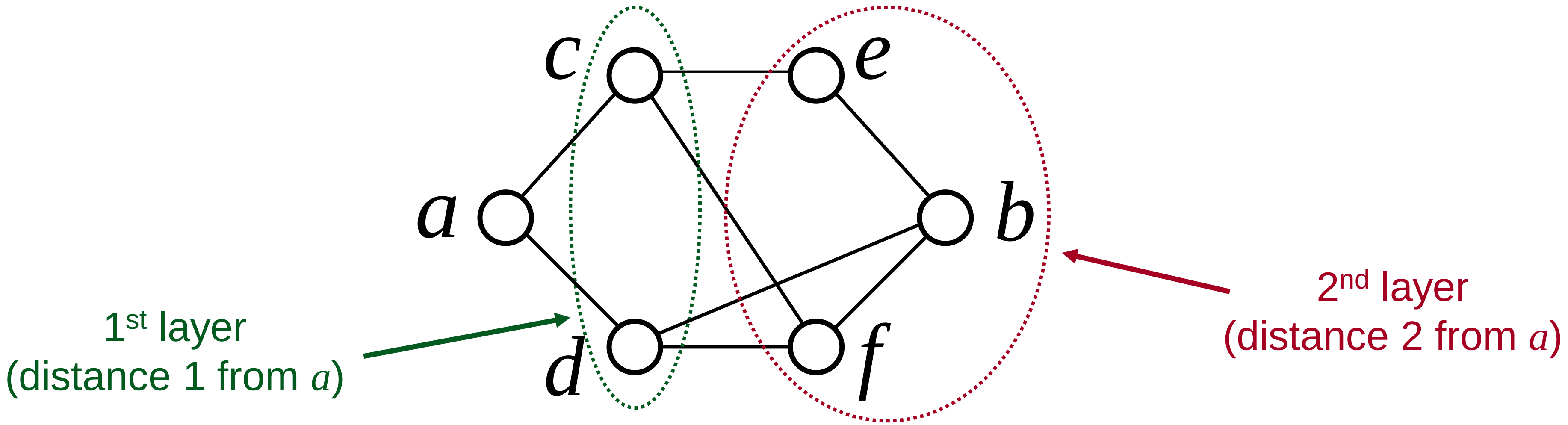
$$\left.\begin{aligned} dist(c,b) &= 2 \\ dist(d,b) &= 1 \end{aligned}\right\} \Rightarrow dist(a,b) = 1+1 \\ = 2$$

path of length 2
(stortest path)

Shortest paths

In this algorithm, we proceed layer-by-layer:

- we expand the “frontier” between visited
  and unvisited vertices, across the breadth of the frontier



- For every $k = 1, 2, 3, \ldots$ the algorithm:
  - first visits all vertices at distance k from a
  - and then all vertices at distance $k+1$

Breadth-First Search (BFS)

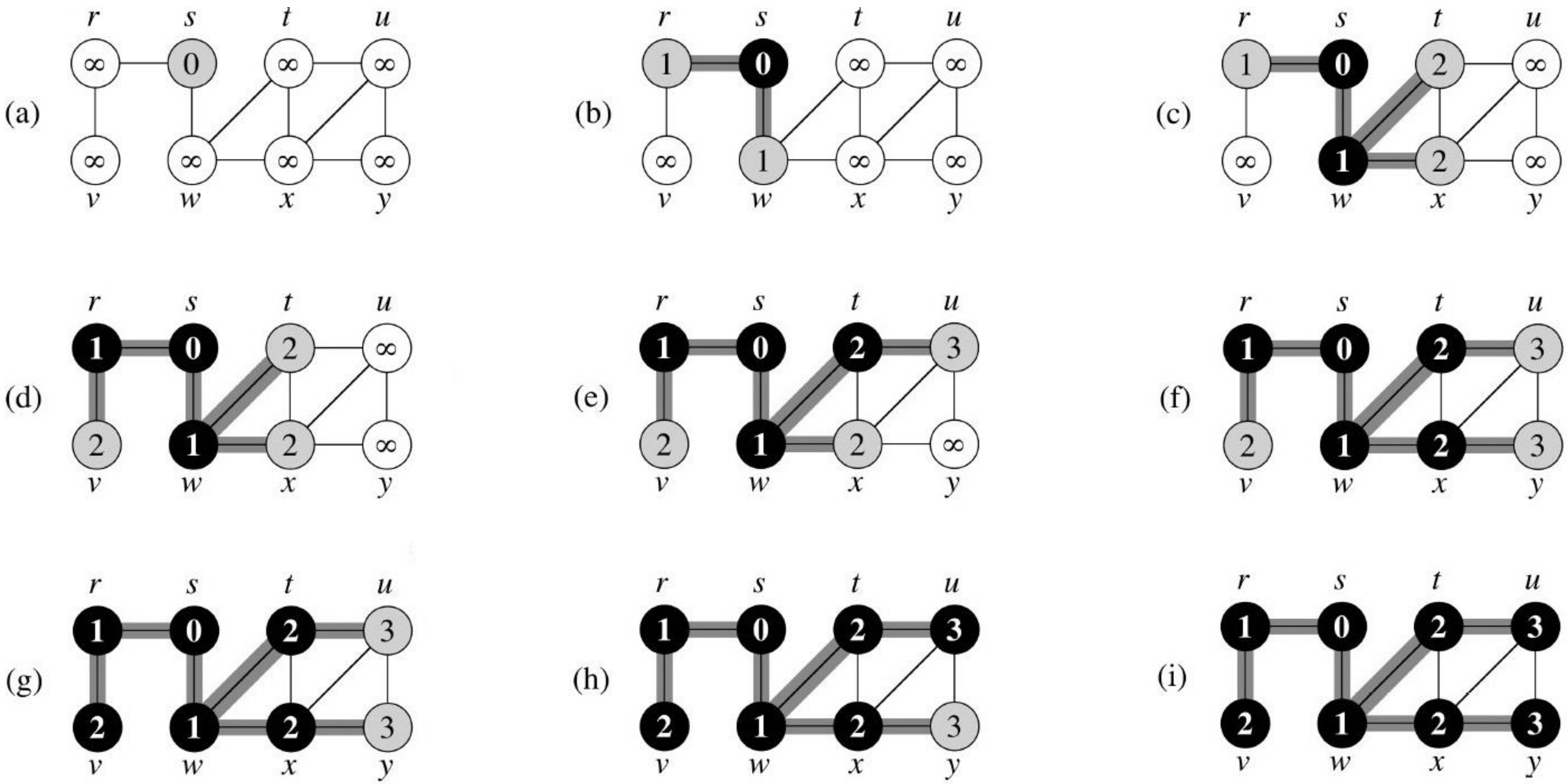
The natural alternative to DFS:

- Breadth-First Search (BFS) algorithm: (pseudo-code)

```
BFS(G, a, b)
i = 0                                              // initialisation
label[a] = 0                                       // initialisation
while b is unlabeled                               // iterate until you reach vertex b
    for each vertex u with label[u] == i
        for each unlabeled vertex v ∈ Adj[u]       // we found a vertex in the next layer
            label[v] = i + 1
    i = i + 1                                      // we increase the counter of the layers
return label[b]
```

- BFS is an iterative algorithm, i.e. no recursive calls
- the label of a vertex u equals its distance from a
- we could continue iteration until all vertices are labelled
- initially all vertices are marked as "unlabeled"
  - i.e. $label[u] = -1$ (or $label[u] = \infty$) for all vertices u

BFS in action

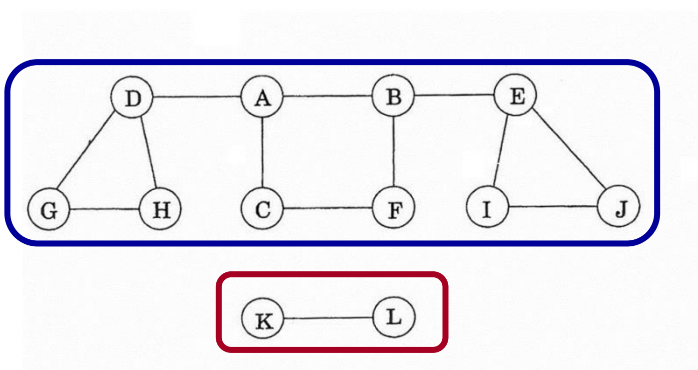


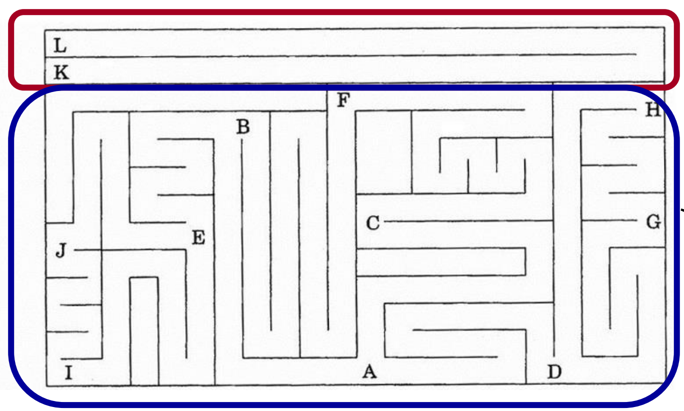
- white vertex: unlabeled
- gray vertex: labelled, but not all its neighbours are labelled
- black vertex: labelled, and all its neighbours are labelled

Graph traversing

- Both data structures:
  - store only “local” information about the graph (i.e. adjacencies)
  - the “global” information is provided implicitly
- How can you know if the graph is connected?
  - if you start at a specific vertex, can you reach every other vertex?
  - if not, can you list the “reachable” vertices?

L is connected only to K





all others are connected to each other

- It is like exploring a labyrinth (maze):
  - can you find a way from vertex D to vertex J ?
  - from A to F ?

Graph traversing

- Where is the difficulty?
  - we want to visit all accessible vertices
  - but avoid running into “cycles”
- An ancient algorithm to traverse a labyrinth: *(Ariadne's string)*
  - whenever you find an unvisited vertex,
    continue to explore from it deeper
  - if no more options, use a *ball of string* to return to junctions:
    - that you previously saw
    - but you did not yet investigate
- How can we do this in a graph?
  - using recursion

Depth-First Search (DFS)

The Depth-First Search (DFS) algorithm: (pseudo-code)

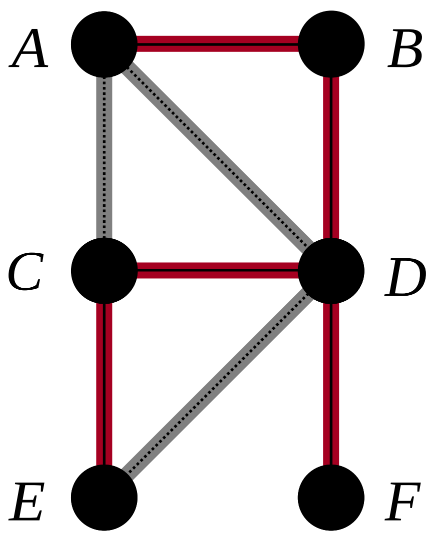
```
DFS(G, u)
1. visited[u] = 1                      // mark u as "visited"
2. print u                             // print vertex u
3. for each vertex v ∈ Adj[u]
4.       if visited[v] == 0 then       // an unvisited vertex v has been discovered
5.             DFS(G, v)               // start (recursively) the DFS search from v
```

recursive call of DFS

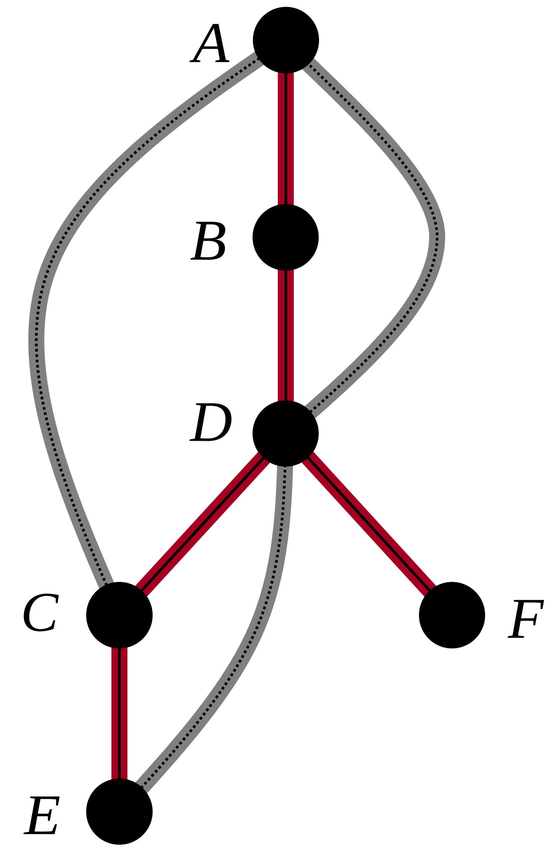
- initially all vertices are marked as "unvisited"
  - i.e. $visited[u] = 0$ for all vertices u
- when we visit a new vertex u:
  - we mark it as "visited" (line 1)
  - we call (recursively) the same algorithm (DFS) for all unvisited neighbours v of u (lines 3 – 5)

DFS in action

The graph:



The DFS-traversal schematically:



The algorithm runs in linear time
(one operation for each vertex and edge)

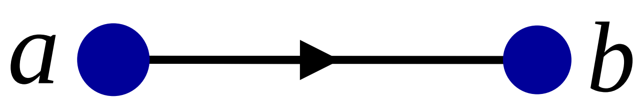
A DFS-visiting order of the vertices: A, B, D, C, E, F

Depth-First Search (DFS)

- The Depth-First Search (DFS) algorithm can be used to traverse the whole graph

- A simulation of DFS in traversing a maze: [here]()

- It can be also used for directed graphs:
  - in this case, $Adj[u]$ denotes the set of vertices that are accessible from u with one edge
  - for example:



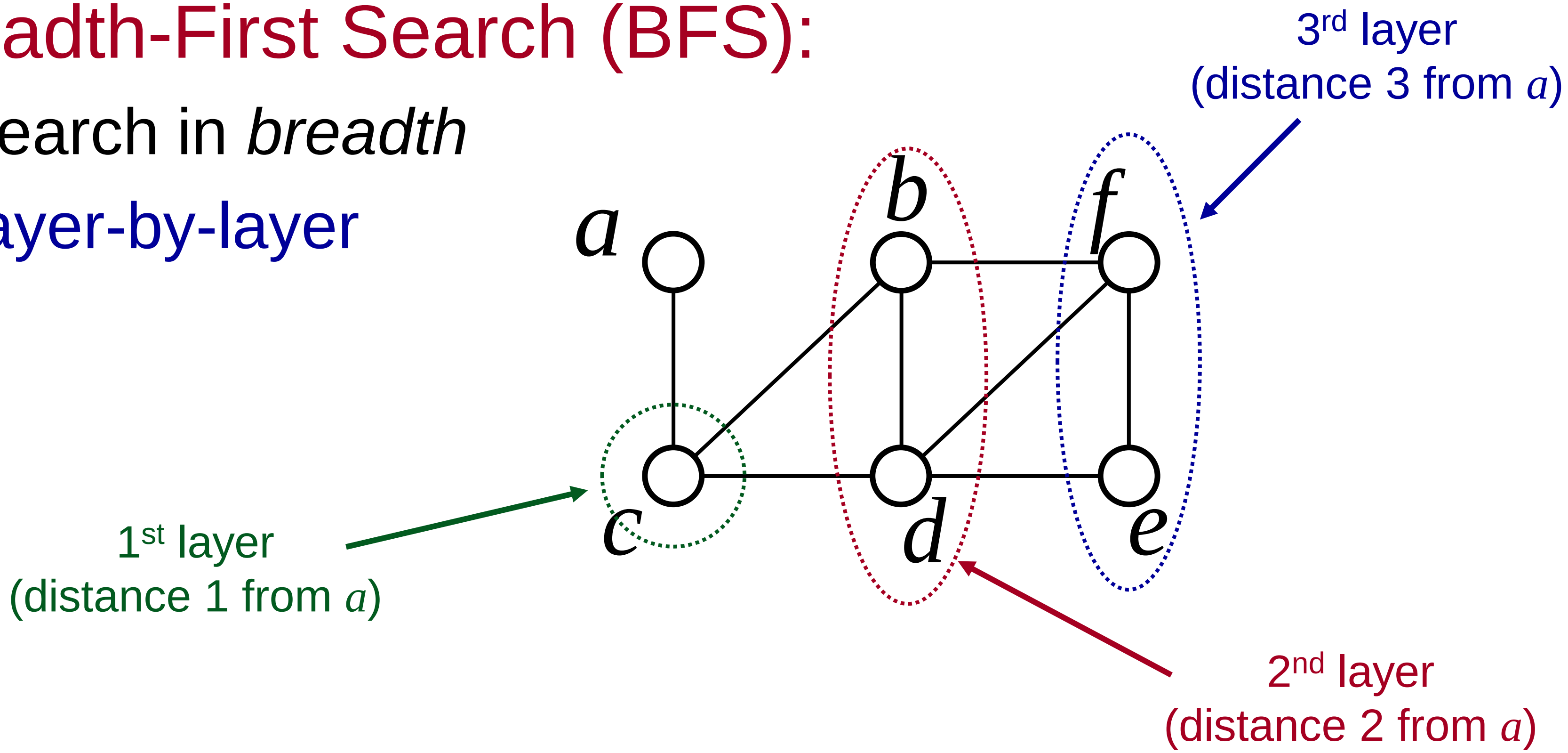
$$b \in Adj[a]$$

$$a \notin Adj[b]$$

BFS vs DFS

Two main approaches for graph exploration:

- Breadth-First Search (BFS):
  - search in *breadth*
  - layer-by-layer



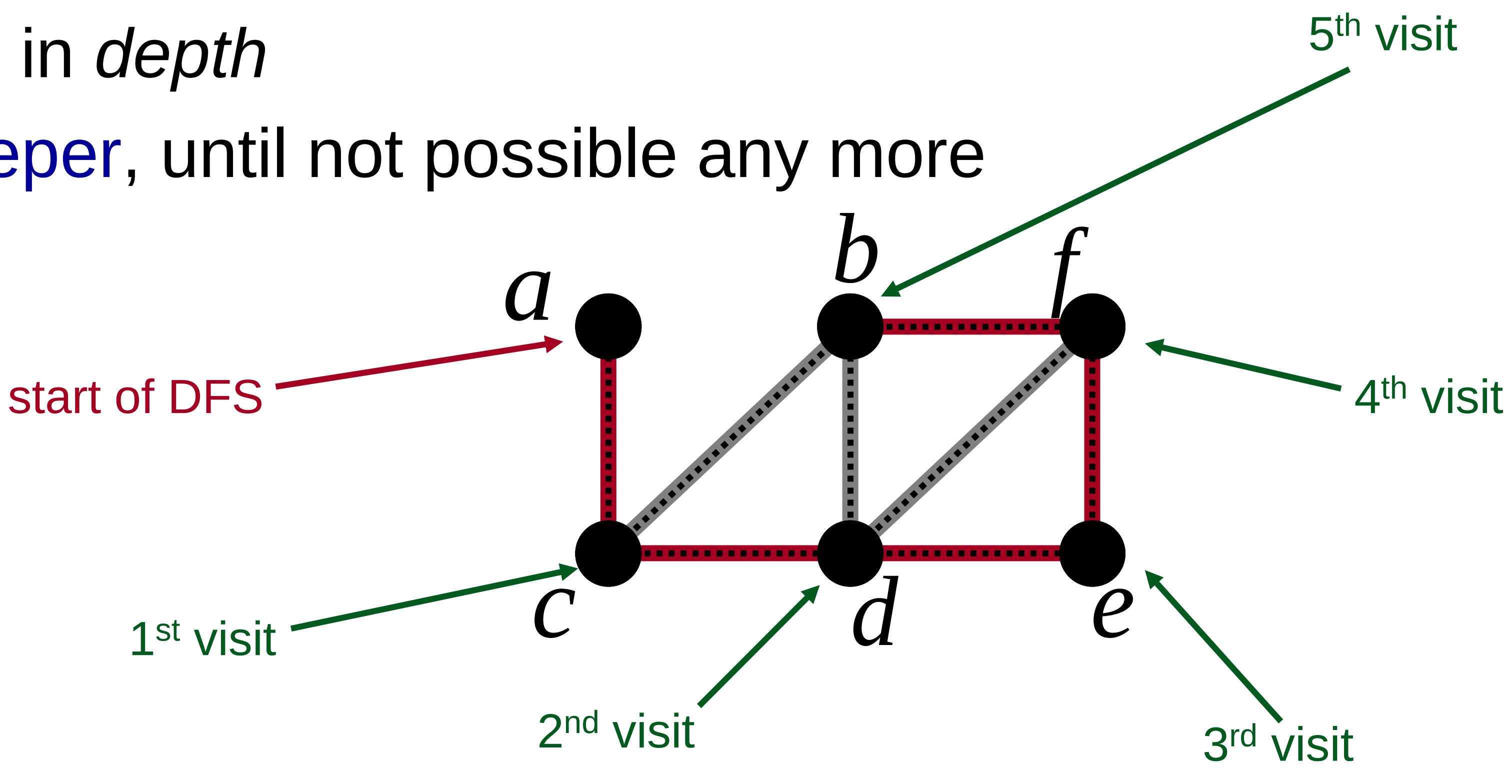
BFS computes shortest paths:

- a and b are at distance 2
- a and f are at distance 3

BFS vs DFS

Two main approaches for graph exploration:

- Depth-First Search (DFS):
  - search in *depth*
  - dig deeper, until not possible any more



DFS reaches b with a path of length 5

- much more than the shortest path !

BFS vs DFS

- DFS is not appropriate for shortest paths:
  - we may reach the target vertex b via a very long path, as we just “dig deeper”
- both BFS and DFS:
  - appropriate for graph exploration
  - can list all reachable vertices from a start vertex a
  - very fast (linear time)
- what else do they have in common?

A generic search algorithm

Generic-Graph-Search(G, a)

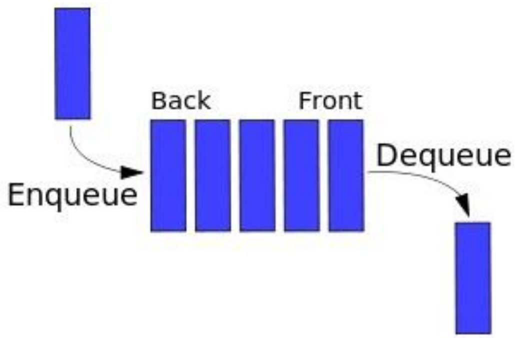
Input: a connected graph G and a vertex a ("source vertex")
Output: an ordered list L of vertices reachable from a

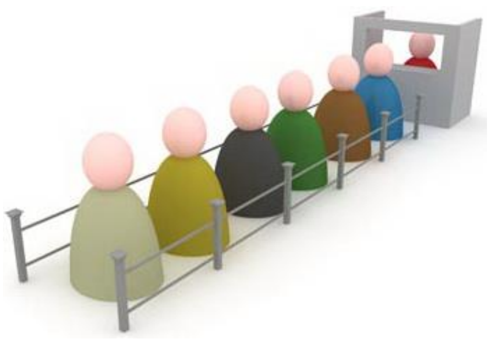
1. $visited[a] = 1$ // initialisation
2. $S = \{a\}$ // initialisation; set of already visited vertices (yet unordered in L), from which we continue exploration
3. $L = []$ // initialisation; ordered list of visited vertices
4. for $i = 1$ to n // iterate until we order all vertices in L
5. pick and remove a vertex $u \in S$ // the crucial choice of the search
6. append L with u // u is the next vertex in the output list
7. for each vertex $v \in Adj[u]$
8. if $visited[v] == 0$ then // we found a new vertex v to reach
9. $visited[v] = 1$ // "mark" v as visited
10. $S = S \cup \{v\}$ // add v to the set S of visited vertices (i.e. yet unordered in L)

- the set S changes dynamically
- BFS and DFS:
  - have different "policy" for the choice at line 4
- BFS prefers vertices "closer to a"
- DFS prefers vertices that are always "one step further"

The policy of BFS

- The policy of BFS:
  - remove the element that has been longer in S
  - a First-In-First-Out (FIFO) policy
- This data-structure is called "queue":

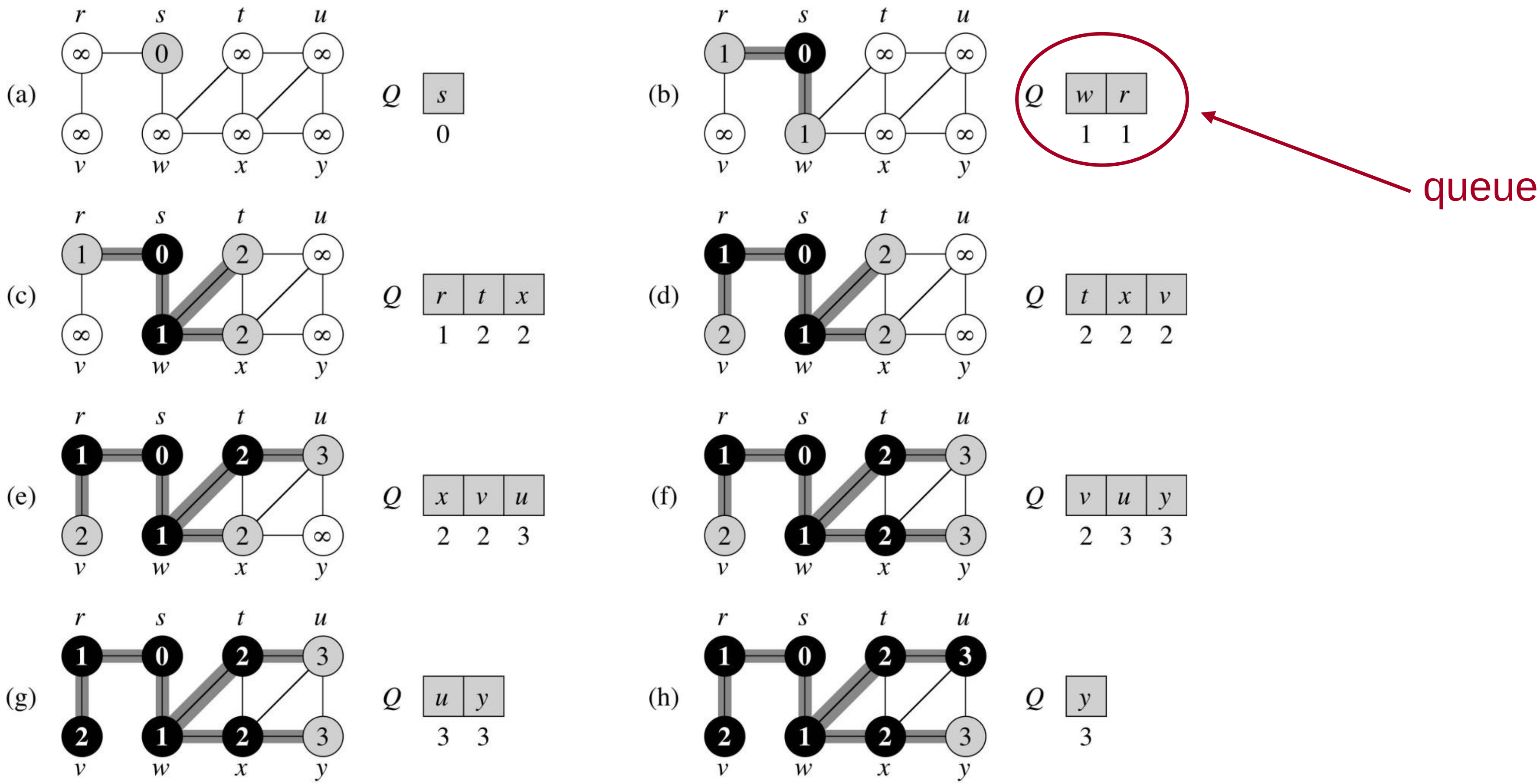


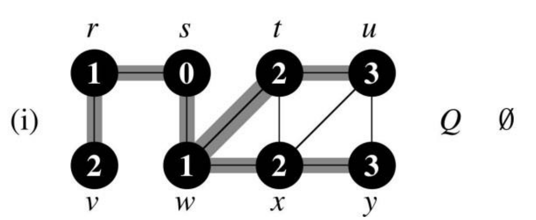


- In other words:
  - add new vertices at the end of the queue
  - remove vertices from the beginning of the queue

$\Longrightarrow$ first process vertices that are closer to the start vertex

BFS example - revisited

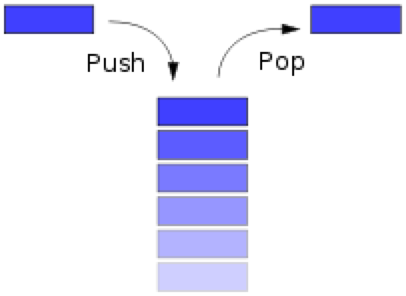




- We implemented (implicitly) the queue using the labels on the vertices

The policy of DFS

- The policy of DFS:
  - remove the element that has been shorter in S
  - a Last-In-First-Out (LIFO) policy
- This data-structure is called "stack":

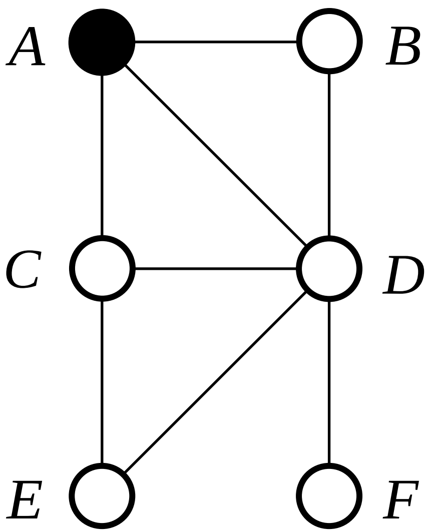


- In other words:
  - add new vertices at the end (top) of the stack
  - remove vertices also from the end (top) of the stack

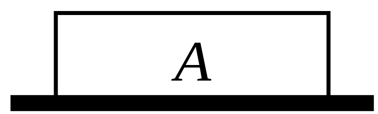
$\Longrightarrow$ first process vertices that always "one step further"

DFS example - revisited

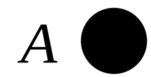
The graph:



The stack:



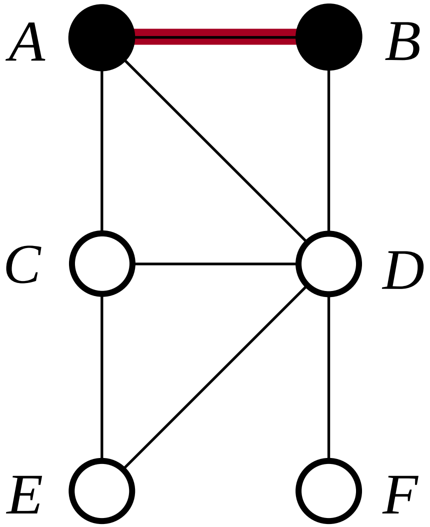
The DFS-traversal schematically:



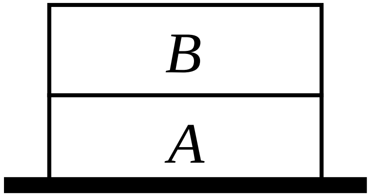
A DFS-visiting order of the vertices: A,

DFS example - revisited

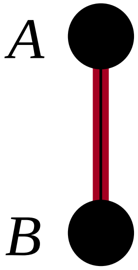
The graph:



The stack:



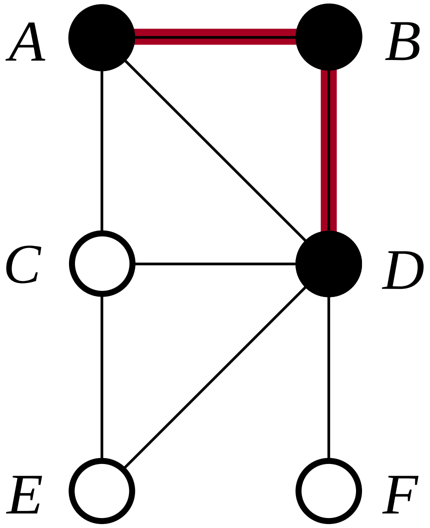
The DFS-traversal schematically:



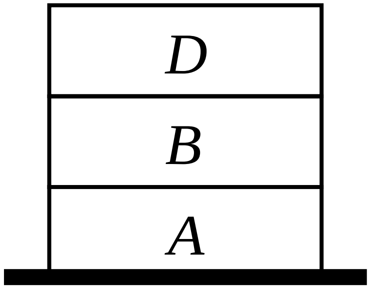
A DFS-visiting order of the vertices: $A, B,$

DFS example - revisited

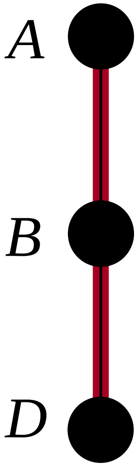
The graph:



The stack:



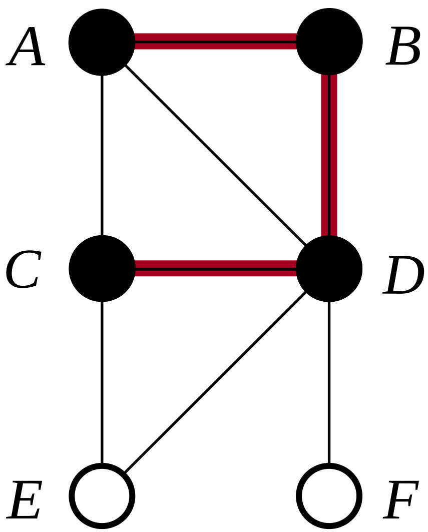
The DFS-traversal schematically:



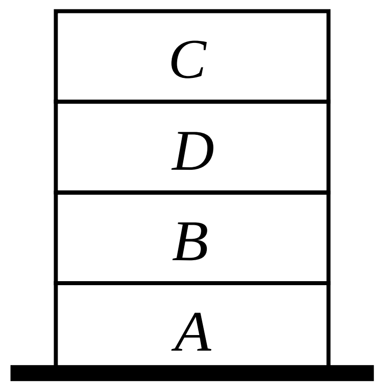
A DFS-visiting order of the vertices: A, B, D,

DFS example - revisited

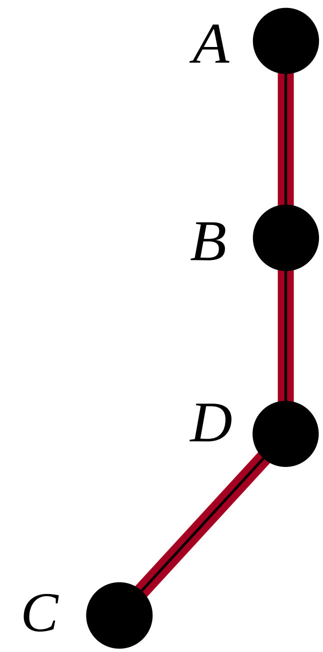
The graph:



The stack:



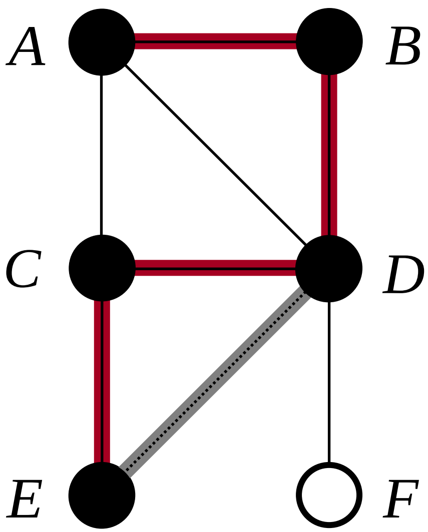
The DFS-traversal schematically:



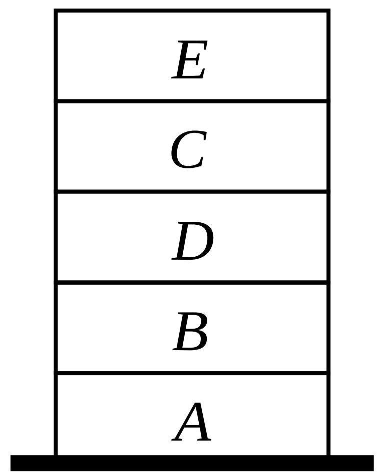
A DFS-visiting order of the vertices: *A*, *B*, *D*, *C*,

DFS example - revisited

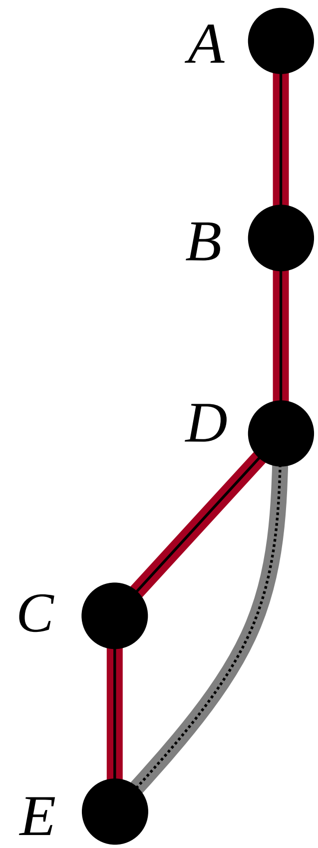
The graph:



The stack:



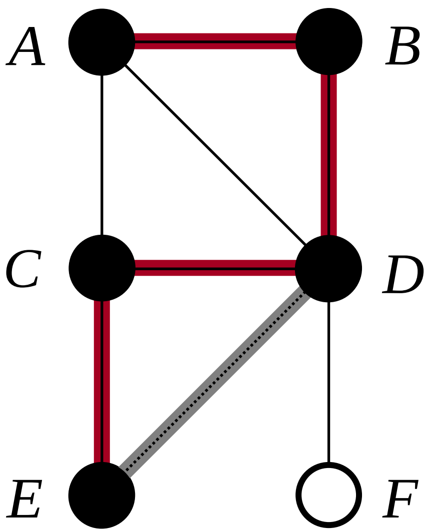
The DFS-traversal schematically:



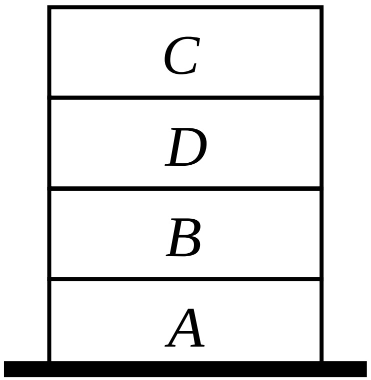
A DFS-visiting order of the vertices: A, B, D, C, E,

DFS example - revisited

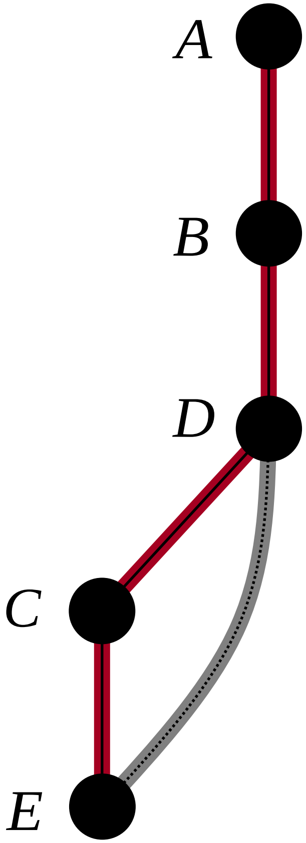
The graph:



The stack:



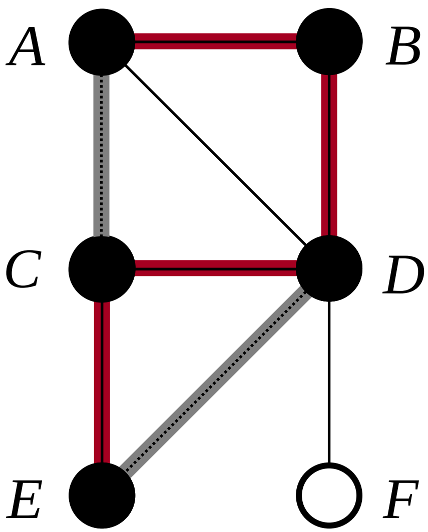
The DFS-traversal schematically:



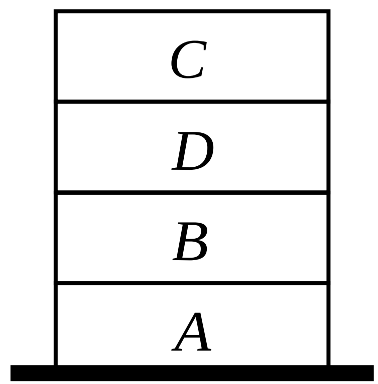
A DFS-visiting order of the vertices: A, B, D, C, E,

DFS example - revisited

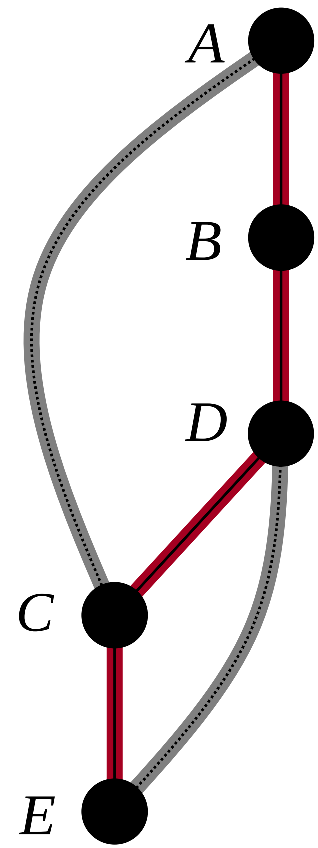
The graph:



The stack:



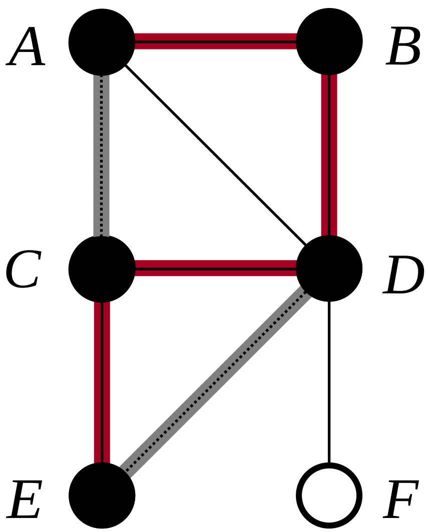
The DFS-traversal schematically:



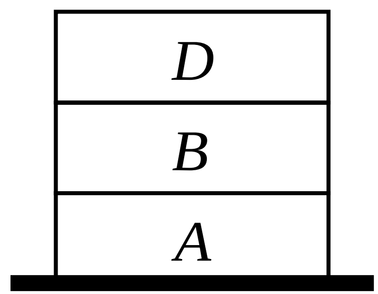
A DFS-visiting order of the vertices: $A, B, D, C, E,$

DFS example - revisited

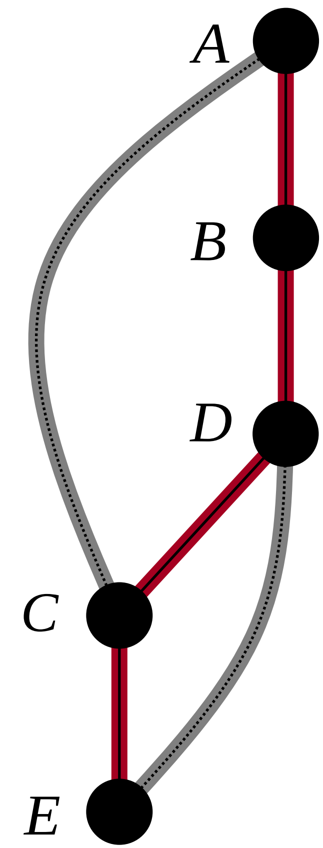
The graph:



The stack:



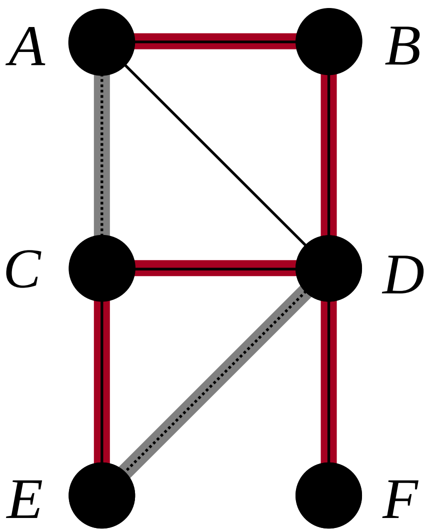
The DFS-traversal schematically:



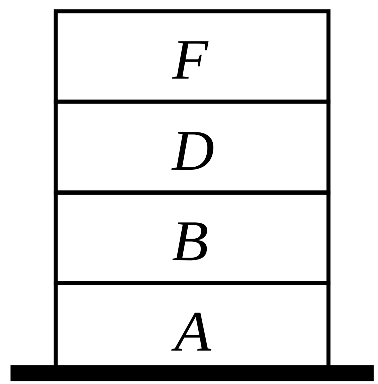
A DFS-visiting order of the vertices: *A*, *B*, *D*, *C*, *E*,

DFS example - revisited

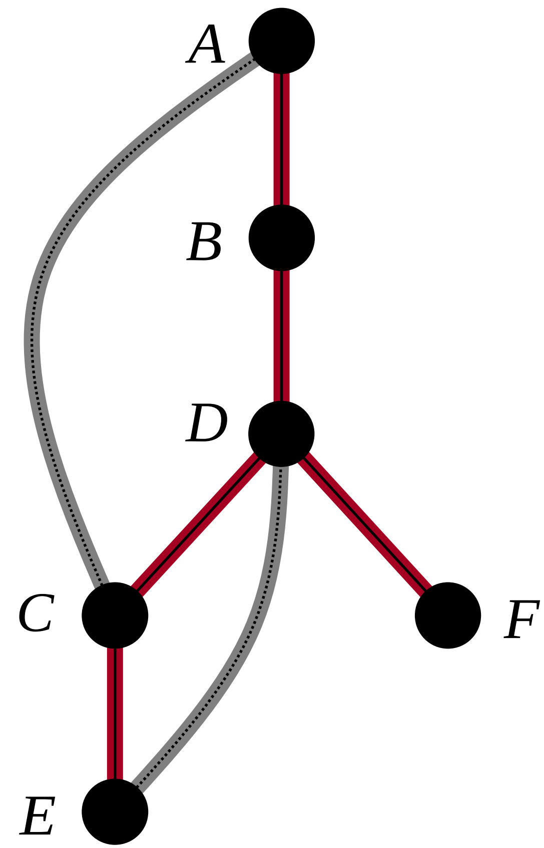
The graph:



The stack:



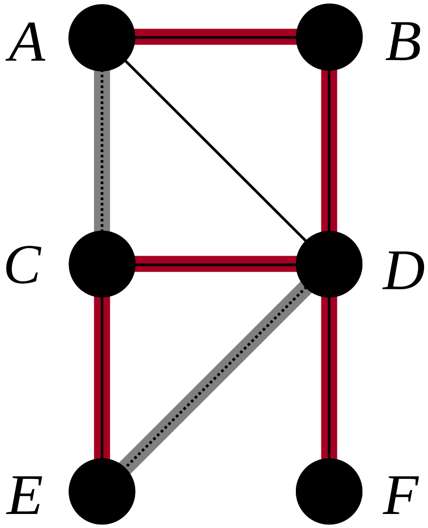
The DFS-traversal schematically:



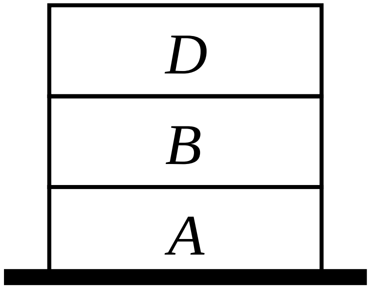
A DFS-visiting order of the vertices: A, B, D, C, E, F

DFS example - revisited

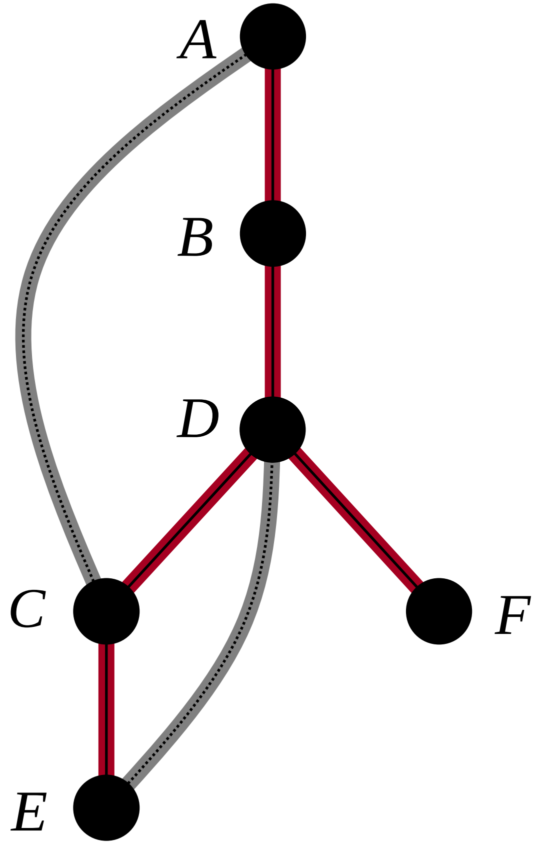
The graph:



The stack:



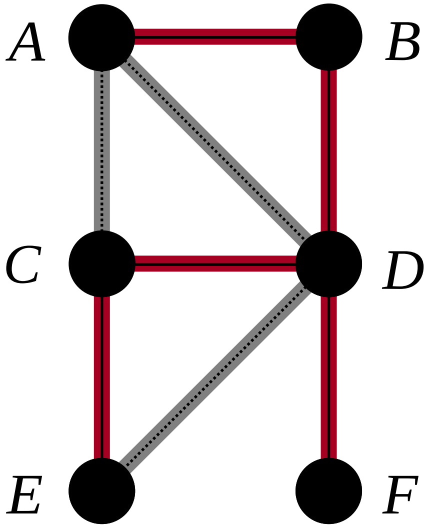
The DFS-traversal schematically:



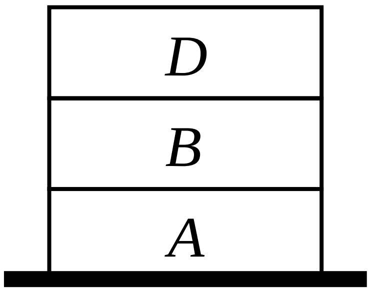
A DFS-visiting order of the vertices: A, B, D, C, E, F

DFS example - revisited

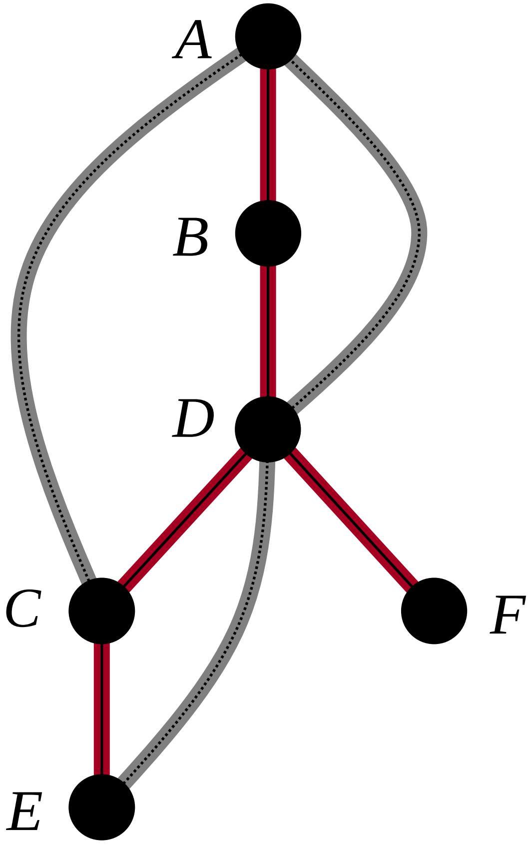
The graph:



The stack:



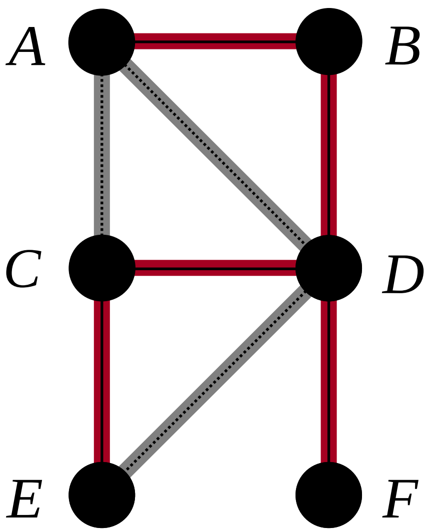
The DFS-traversal schematically:



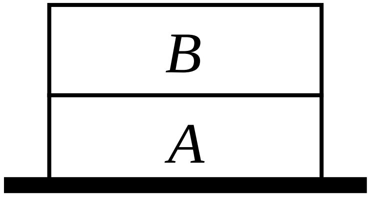
A DFS-visiting order of the vertices: A, B, D, C, E, F

DFS example - revisited

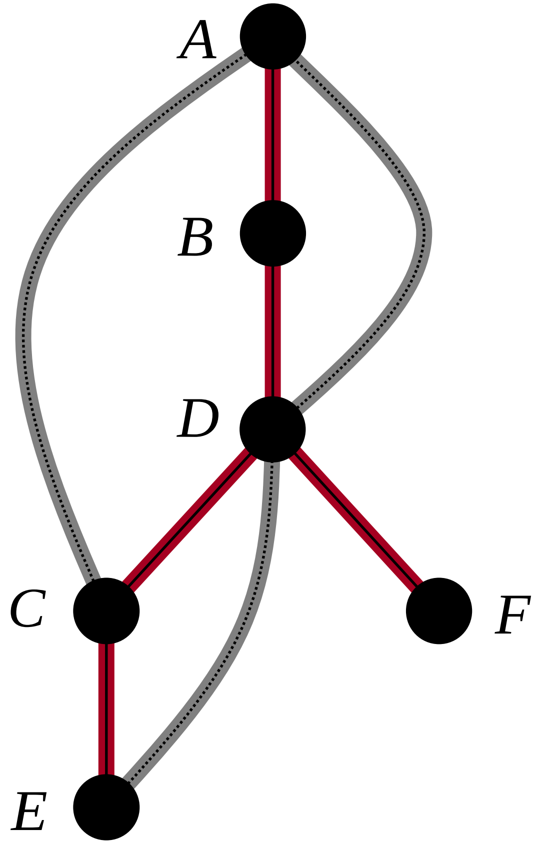
The graph:



The stack:



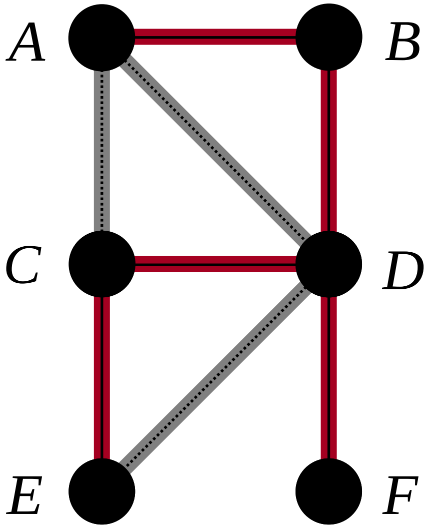
The DFS-traversal schematically:



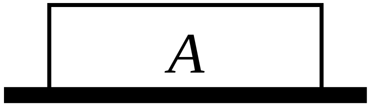
A DFS-visiting order of the vertices: A, B, D, C, E, F

DFS example - revisited

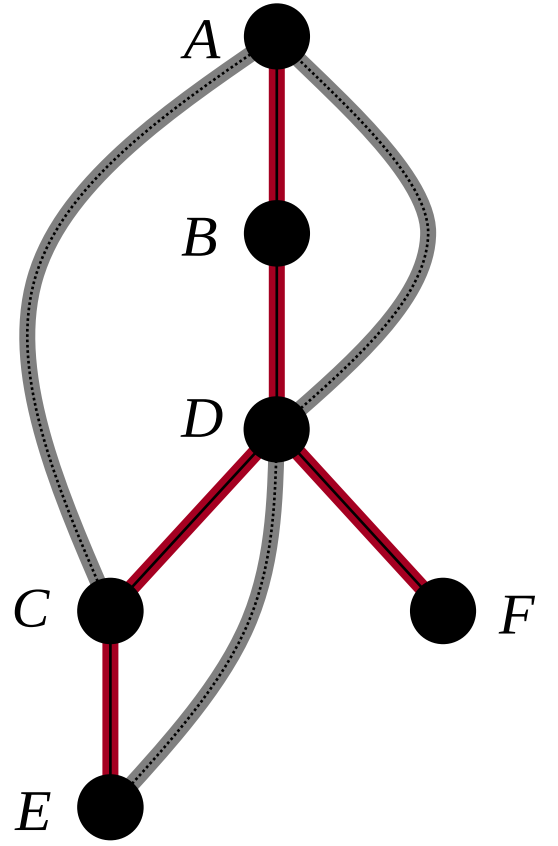
The graph:



The stack:



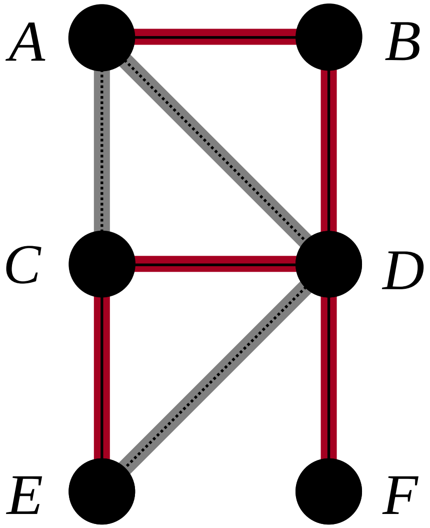
The DFS-traversal schematically:



A DFS-visiting order of the vertices: A, B, D, C, E, F

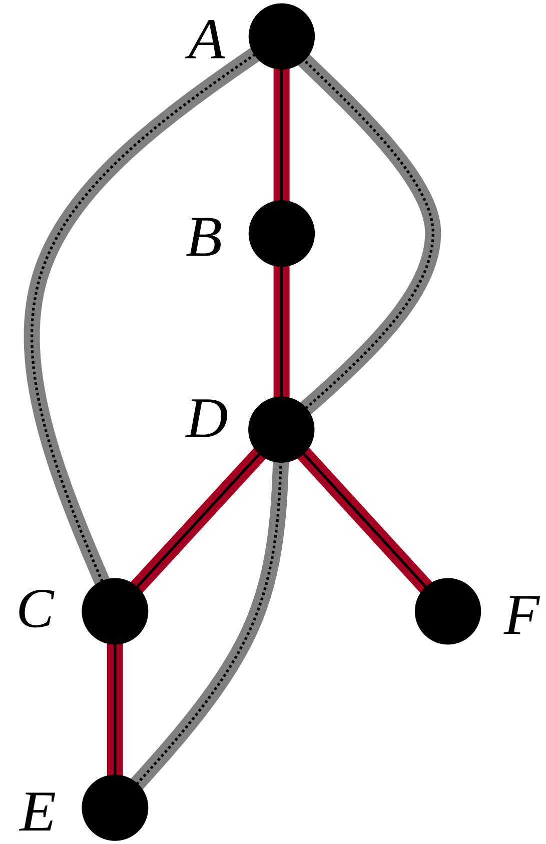
DFS example - revisited

The graph:



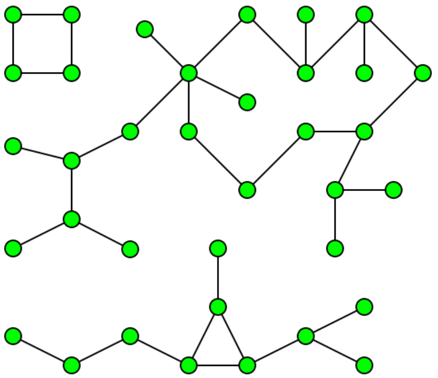
The stack:

The DFS-traversal schematically:



A DFS-visiting order of the vertices: A, B, D, C, E, F

Graph traversing



- Variations of DFS are mainly used for "connectivity-type" problems, e.g.
  - to find all connected components
  - solve "reachability" problems

Other practical applications:

You are the mayor of a small town.
An unholy coalition of shop owners, who want more street-side parking, proposes to turn most streets into one-way streets.
You want to ensure that in their new plan, one can still drive from any point in town to any other point.

How can you check that with DFS?

- "strongly connected" directed graphs

Longest paths

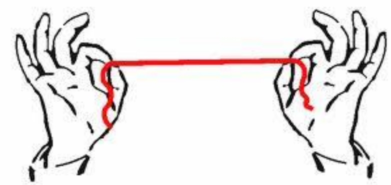
- Ok, we know how to compute efficiently
  a shortest path between two given vertices (BFS)
- What about computing a longest path?
  - superficially similar problems
  - however very different !

Theorem: *it is NP-complete to compute a longest path between two given vertices.*

- In other words:
  - *"nobody knows any efficient algorithm that always computes a longest path"*

Longest paths

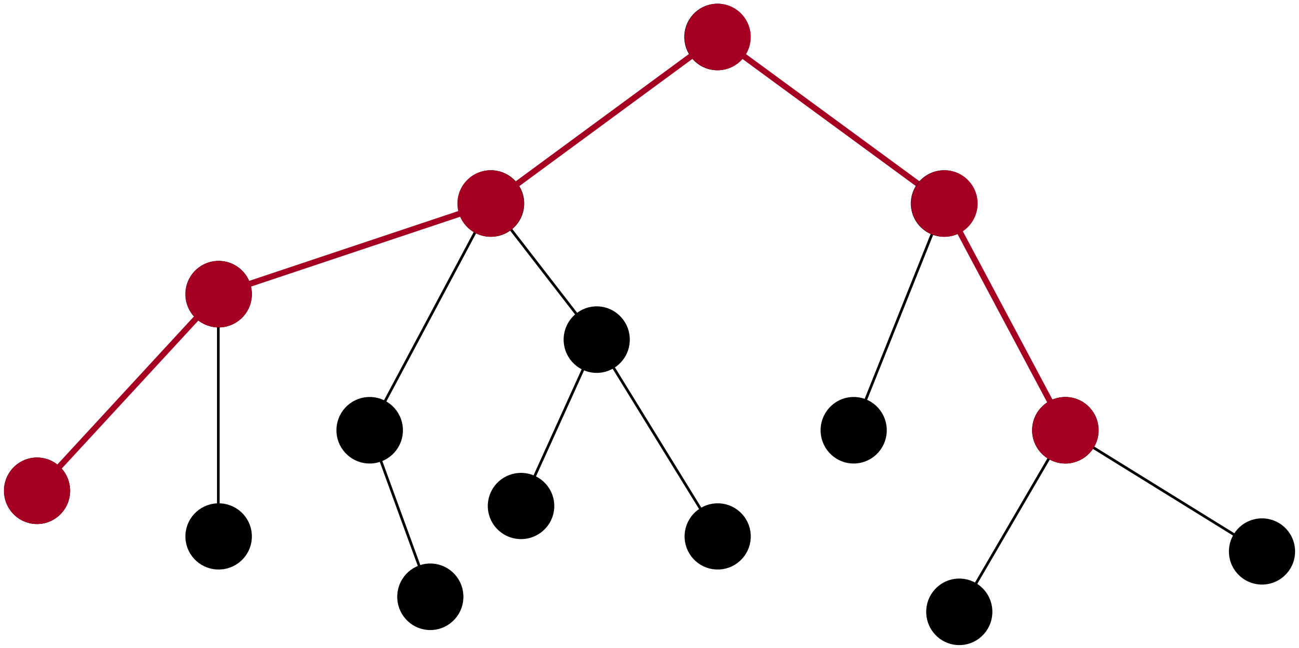
- Intuitively:
  - vertices are balls
  - edges are strings tight on the balls
- shortest path problem:
  - pull firmly two specific balls away from each other
  - the length of the string between them is their distance in the graph

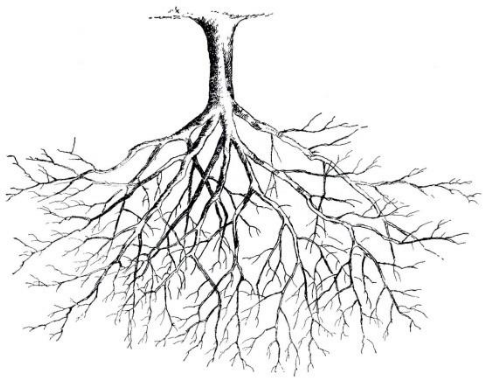


- longest path problem:
  - you need to investigate all (possibly "strange") paths between the two balls through the net of strings
  - which can be very complex

Longest paths

- However:
  - if the graph has no cycles, then it is easy
  - such graphs are called “trees”

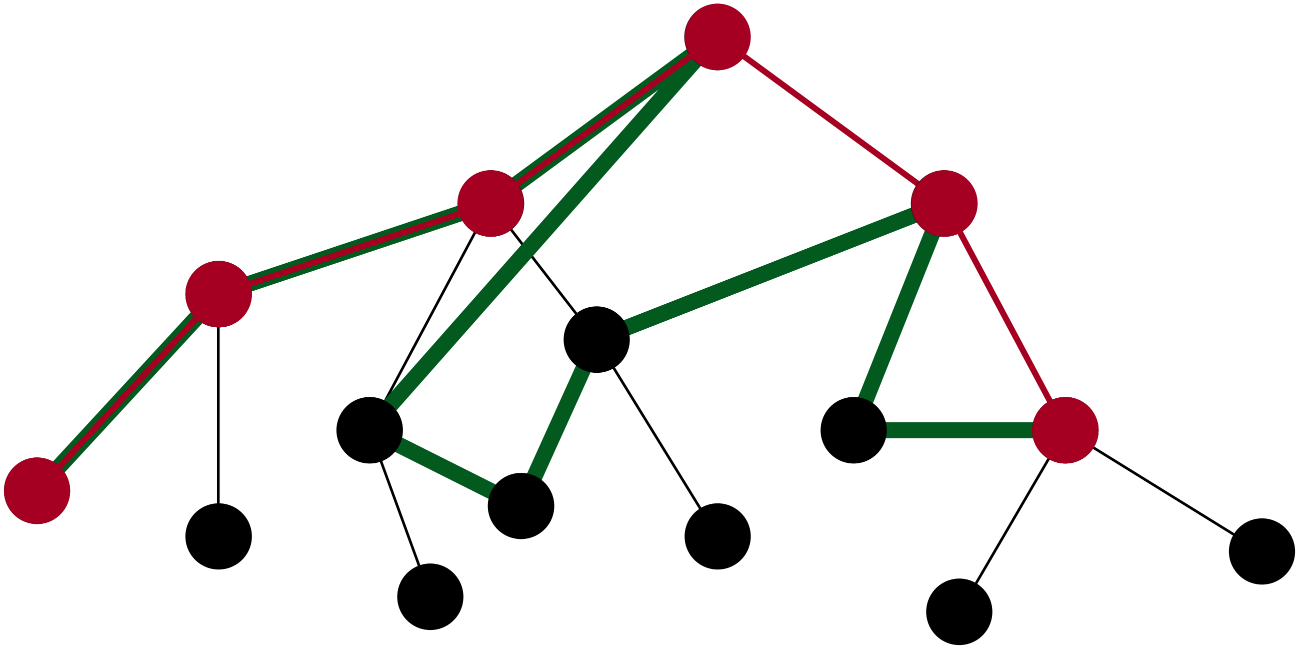


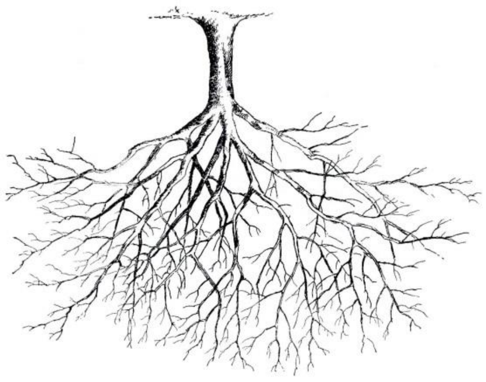


- Any pair of vertices is connected with exactly one path !

Longest paths

- However:
  - if the graph has no cycles, then it is easy
  - such graphs are called “trees”





- What can happen if we have cycles?