Java II
Graphs and Search

Bernd Kiefer

Deutsches Forschungszentrum für künstliche Intelligenz

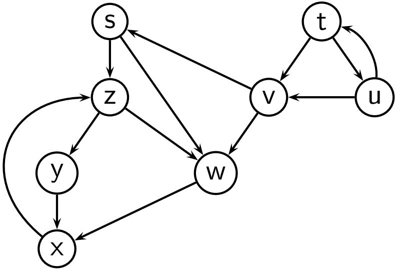
Graphs: Definition

- Graph $\mathcal{G}$: A set of vertices (nodes) $\mathcal{V}$ and a set of edges $\mathcal{E}$, which is a relation on vertices, that is: $\mathcal{E} \subseteq \mathcal{V} \times \mathcal{V}$
- Example:
  - Vertices: students at the university
  - $(u, v) \in \mathcal{E} \Leftrightarrow$ student u knows student v
- Graphical representation:
  - vertices: blobs
  - edges: arrows (arcs) between the blobs
- If $\mathcal{E}$ is symmetric, i.e., if $(u, v) \in \mathcal{E} \Leftrightarrow (v, u) \in \mathcal{E}$ the graph is called *undirected* (plain arcs, not arrows)
- Example: $\mathcal{E}$ is the set of students that are akin

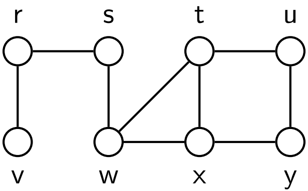
Graphs: Definitions II

- Vertex u is *reachable* from vertex v ($u \rightarrow v$) iff there is a sequence of edges $(u, w_1), (w_1, w_2), \ldots, (w_n, v)$ in $\mathcal{E}$
- A graph is *cyclic* (contains a cycle) if there is $u \in \mathcal{V}$ s.th. $u \rightarrow u$ over a nontrivial sequence of edges in $\mathcal{E}$, including a self loop

directed graph



undirected graph



Implementation Basics

- Represent the vertices as numbers from zero to $|\mathcal{V}| - 1$
- Matrix representation: represent $\mathcal{E}$ as a quadratic `boolean` matrix A of size $|\mathcal{V}|$; $A[i, j]$ is `true` iff $(i, j) \in \mathcal{E}$

+ Good for dense graphs, where $|\mathcal{E}| \approx |\mathcal{V}|^2$: only one bit per edge
+ Fast: are two vertices directly connected?

– Initialization is quadratic in $|\mathcal{V}|$
– Visiting all outgoing edges of a vertex takes $|\mathcal{V}|$ steps, no matter how many there really are
– Additional information attached to the edges (e.g., weights) has to be stored separately

Adjacency List Representation

- For every vertex, store a list of outgoing edges, i.e., the vertex number that is reached
- Graph is represented by an array of list heads
- In Java: `ArrayList` of `Lists`.
- \+ Compact representation for most graphs, except if they are very dense
- \+ Allows more efficient implementations of many graph algorithms
- \+ Additional edge information can be stored in the elements of the edge lists directly

Search in Graphs

- Task: visit all reachable vertices, starting at vertex s
- Iteratively use all the outgoing edges of s, and all the nodes that can be reached through these egdes
- Make sure that no node gets explored twice
- Basic idea: maintain two sets
  - $\mathcal{U}$ the *visited* nodes
  - $\mathcal{A}$ the *active* nodes, i.e., still unexplored outedges
- In textbooks, vertices are often assigned colors during the search:
  - White: not in $\mathcal{U}$ and not in $\mathcal{A}$
  - Grey: in $\mathcal{U}$ and in $\mathcal{A}$ (under consideration)
  - Black: in $\mathcal{U}$, but not in $\mathcal{A}$ anymore (finished)

Generic Search Algorithm

- Initialization: both sets contain only the start vertex s

proc *Search*(s) $\equiv$
  $\mathcal{U} = \mathcal{A} = \{s\}$ // s gets grey
  while $\mathcal{A} \neq \emptyset$ **do**
    for some node $n \in \mathcal{A}$
      if there is an unused edge $e = (n, m)$ leaving n
        if $m \notin \mathcal{U}$ **then** // m gets grey
          $\mathcal{U} = \mathcal{U} \cup \{m\}$; $\mathcal{A} = \mathcal{A} \cup \{m\}$
      else
        $\mathcal{A} = \mathcal{A} - \{n\}$ // n gets black

- Questions:
  - How to implement sets $\mathcal{U}$ and $\mathcal{A}$?
  - Does the result depend on the implementation?

Implementation of $\mathcal{U}$

- What is the best data structure for $\mathcal{U}$?
- What are the operations on $\mathcal{U}$?

Implementation of $\mathcal{U}$

- What is the best data structure for $\mathcal{U}$?
- What are the operations on $\mathcal{U}$?
  1. Add a node m
  2. Is node n contained in the set?

Implementation of $\mathcal{U}$

- What is the best data structure for $\mathcal{U}$?
- What are the operations on $\mathcal{U}$?
  1. Add a node m
  2. Is node n contained in the set?
- $\mathcal{U}$ should be implemented as a bit vector over the nodes
- Two alternatives:
  - `boolean` member variable of the node data structure
  - A so-called *property vector* (or *property map*) attached to the vertices

Property Vectors

Advantages and drawbacks of property vectors

- More flexible:
  - Create all and only those you need for an algorithm
  - In a graph framework, one can not put all the data into the vertices
  - May contain any type, small or bigger datastructures
  - Only use memory when they are needed
- Require an efficient indexing between vertices and values: maintain a numeric index in the vertices
- Member variables are always faster

Property vectors can also be used for graph edges

Implementation of $\mathcal{A}$

The choice of the data structure for $\mathcal{A}$ and the decisions about n and e determine the order in which vertices are visited

- Operations on set $\mathcal{A}$:

Implementation of $\mathcal{A}$

The choice of the data structure for $\mathcal{A}$ and the decisions about n and e determine the order in which vertices are visited

- Operations on set $\mathcal{A}$:
  - Add a vertex
  - Get and remove some vertex (nondeterministic)
  - Test if set is empty

Implementation of $\mathcal{A}$

The choice of the data structure for $\mathcal{A}$ and the decisions about *n* and *e* determine the order in which vertices are visited

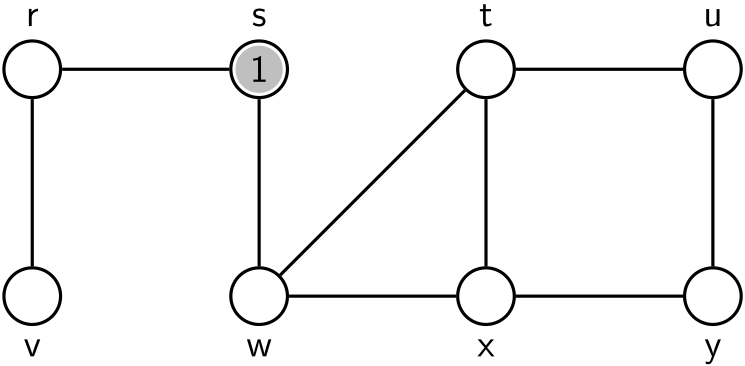
- Operations on set $\mathcal{A}$:
  - Add a vertex
  - Get and remove some vertex (nondeterministic)
  - Test if set is empty
- Implement $\mathcal{A}$ as a *queue* and keep *n* until it gets black: Breadth First Search (BFS)
- Implement $\mathcal{A}$ as a *stack* and always take its top element: Depth First Search (DFS)
- DFS is often implemented as a recursive function, the function call stack takes the role of $\mathcal{A}$

BFS Implementation

```
proc BFS() ≡
  foreach v ∈ 𝒱 do d(v) = 0
  time = 1          // the time when a vertex is touched
  foreach v ∈ 𝒱 with d(v) == 0 do        // v is the start node
    d(v) = time; 𝒜.push_back(v)          // v gets grey
    while ¬ 𝒜.empty() do
        n = 𝒜.pop_front()
        time = d(n) + 1
        foreach e = (n, m) do
          if d(m) == 0 then          // m ∉ 𝒰 ?
            d(m) = time; 𝒜.push_back(m)          // m gets grey
        // n gets black
```

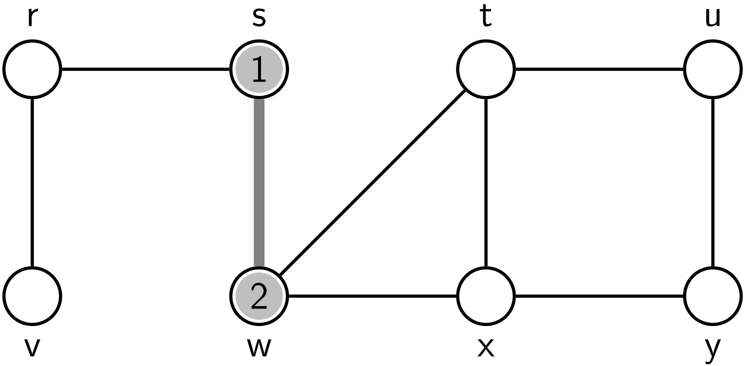
- Finally, all vertices of $\mathcal{G}$ have been visited
- The $d(v)$ is abused to serve as the $\mathcal{U}$ bitvector

Breadth First Search

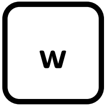


$\mathcal{A} \rightarrow$

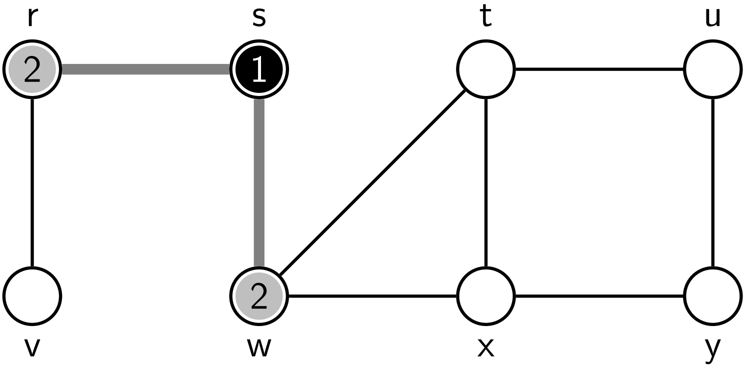
Breadth First Search



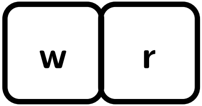
$\mathcal{A} \rightarrow$



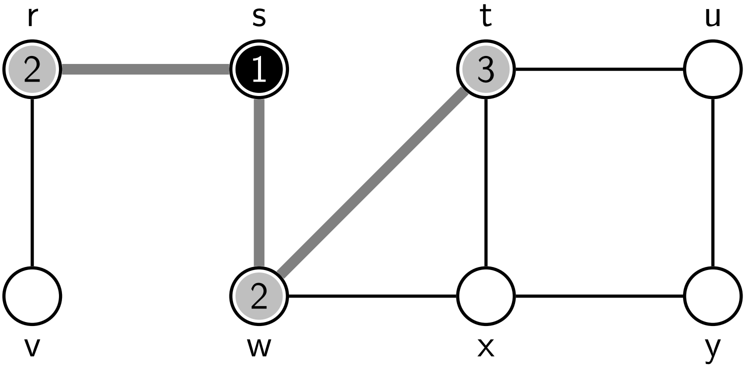
Breadth First Search



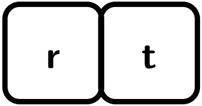
$\mathcal{A} \rightarrow$



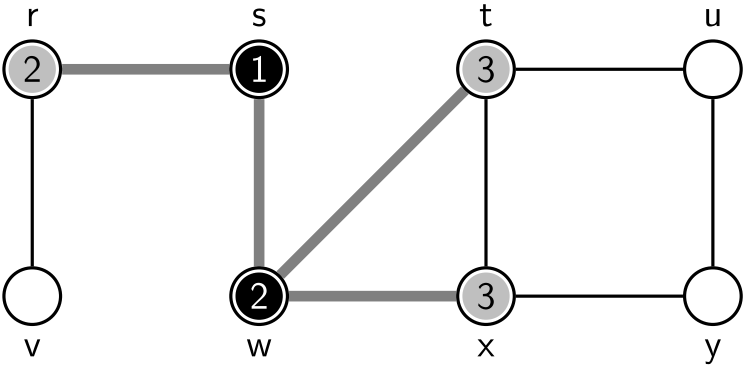
Breadth First Search

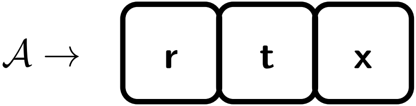


$\mathcal{A} \rightarrow$

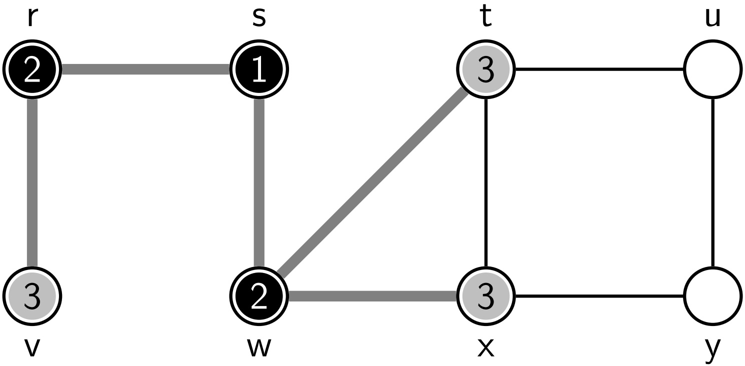


Breadth First Search



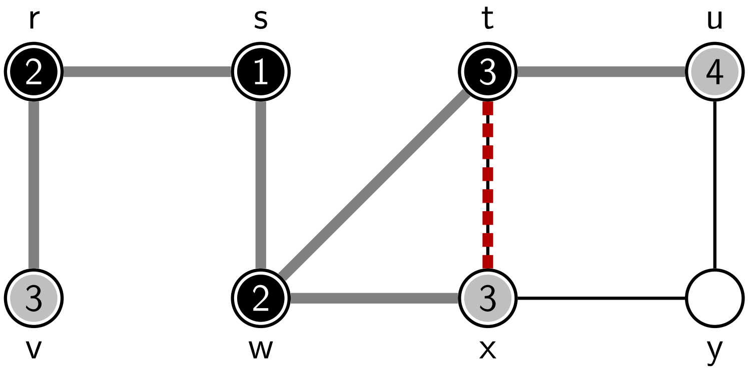


Breadth First Search



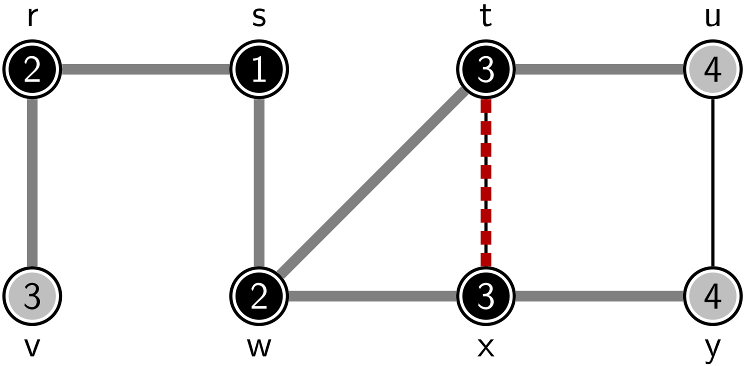
$\mathcal{A} \rightarrow$ | t | x | v |

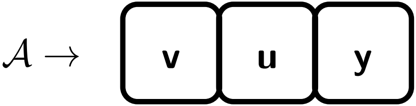
Breadth First Search



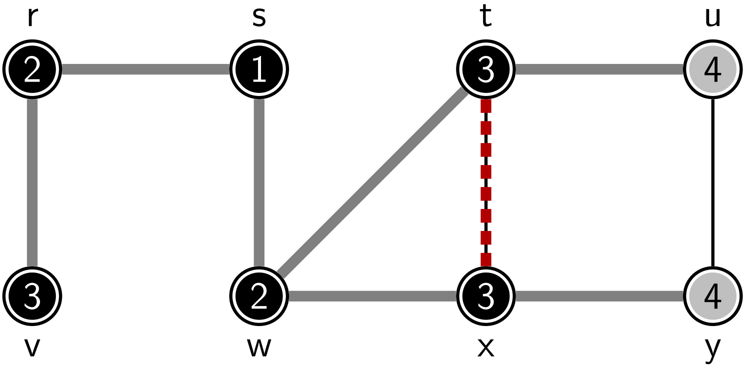
$\mathcal{A} \rightarrow$ | **x** | **v** | **u** |

Breadth First Search



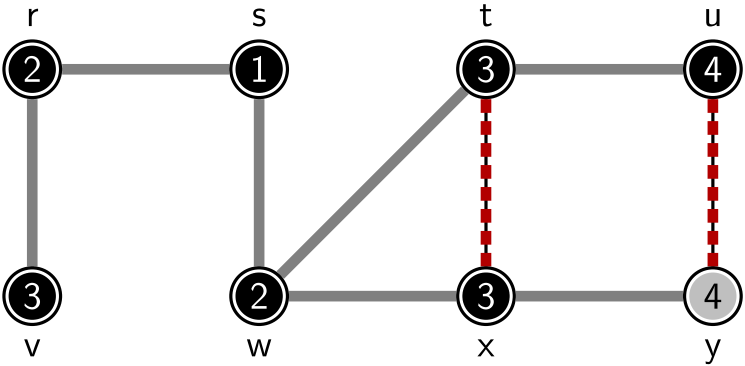


Breadth First Search



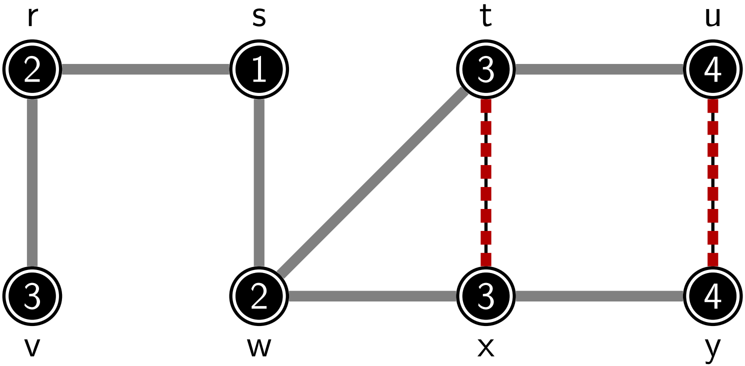
$\mathcal{A} \rightarrow$ **u** **y**

Breadth First Search



$\mathcal{A} \rightarrow$ **y**

Breadth First Search



$\mathcal{A} \rightarrow$

Properties of BFS

- Run-time complexity of BFS?

Properties of BFS

- Run-time complexity of BFS?
- All operations on $d(v)$ and $\mathcal{A}$ need $\mathcal{O}(1)$ time
- The outer loop is traversed $|\mathcal{V}|$ times
- The inner loop touches all edges, so at least $|\mathcal{E}|$ times

Properties of BFS

- Run-time complexity of BFS?
- All operations on $d(v)$ and $\mathcal{A}$ need $\mathcal{O}(1)$ time
- The outer loop is traversed $|\mathcal{V}|$ times
- The inner loop touches all edges, so at least $|\mathcal{E}|$ times

  $\longrightarrow$ overall complexity is $\mathcal{O}(\mathcal{V} + \mathcal{E})$

Properties of BFS

- Run-time complexity of BFS?
- All operations on $d(v)$ and $\mathcal{A}$ need $\mathcal{O}(1)$ time
- The outer loop is traversed $|\mathcal{V}|$ times
- The inner loop touches all edges, so at least $|\mathcal{E}|$ times

  $\longrightarrow$ overall complexity is $\mathcal{O}(\mathcal{V} + \mathcal{E})$
- Grey edges mark first discoveries of neighbor nodes
- They obviously form a tree
- Do you have an interpretation for $d(v)$?

Properties of BFS

- Run-time complexity of BFS?
- All operations on $d(v)$ and $\mathcal{A}$ need $\mathcal{O}(1)$ time
- The outer loop is traversed $|\mathcal{V}|$ times
- The inner loop touches all edges, so at least $|\mathcal{E}|$ times

  $\longrightarrow$ overall complexity is $\mathcal{O}(\mathcal{V} + \mathcal{E})$
- Grey edges mark first discoveries of neighbor nodes
- They obviously form a tree
- Do you have an interpretation for $d(v)$?
- In fact, $d(v) - 1$ is the *minimal distance* from the startnode
- The (grey) tree edges are minimal length paths

DFS: Recursive Procedure

$\underline{\textbf{proc}}\ \mathit{DFS}(G)\ \equiv$
$\quad \underline{\textbf{foreach}}\ v \in \mathcal{V}\ \underline{\textbf{do}}\ d(v) = 0$
$\quad \mathit{time} = 1$ // the time when a vertex is touched
$\quad \underline{\textbf{foreach}}\ v \in \mathcal{V}\ \underline{\textbf{do}}\ \underline{\textbf{if}}\ d(v) == 0\ \underline{\textbf{then}}\ \mathit{DFS\text{-}Visit}(v)$

$\underline{\textbf{proc}}\ \mathit{DFS\text{-}Visit}(v)\ \equiv$
$\quad d(v) = \mathit{time};\ \mathit{time} = \mathit{time} + 1$ // v gets grey
$\quad \underline{\textbf{foreach}}\ e = (v, u)\ \underline{\textbf{do}}$
$\quad\quad \underline{\textbf{if}}\ d(u) == 0\ \underline{\textbf{then}}\ \mathit{DFS\text{-}Visit}(u)$ // is u white? Then visit it.
$\quad f(v) = \mathit{time};\ \mathit{time} = \mathit{time} + 1$ // v gets black

We store two timestamps for each vertex v

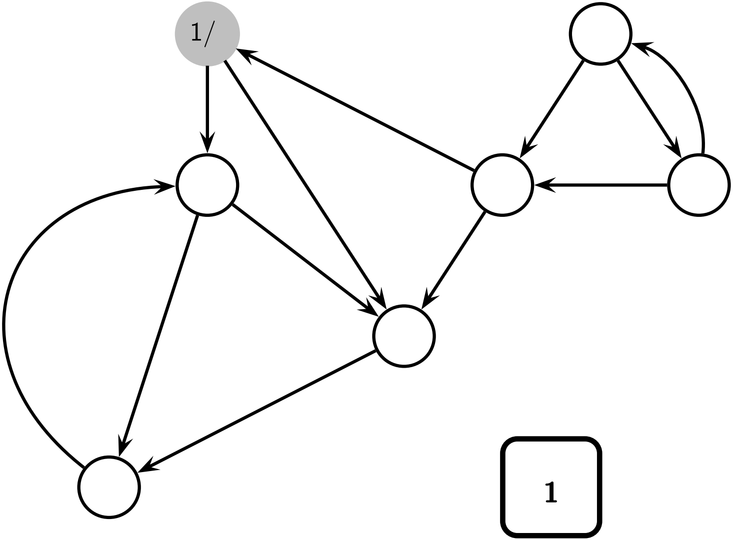
- the *discovery time* $d(v)$, when v changes from white to grey
- the *finishing time* $f(v)$, when v changes from grey to black

Edge Classification using DFS

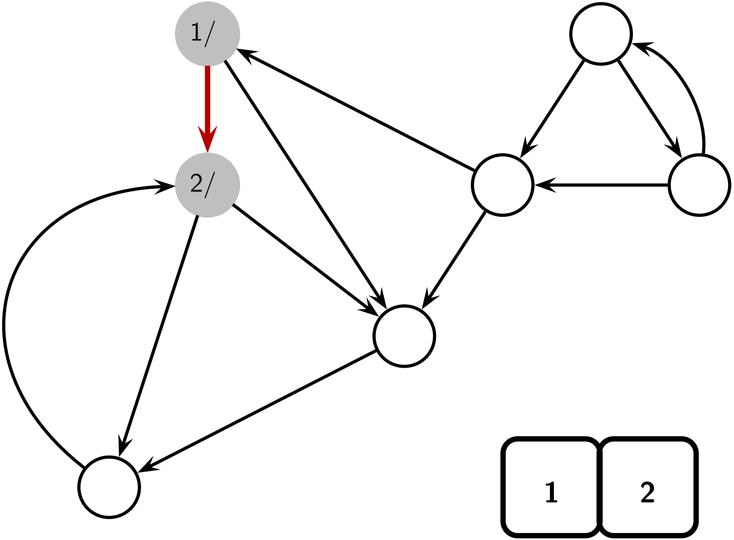
The edges of a directed graph can be classified into four categories, depending on the role they play in a run of depth first search.

- tree edges: the edges used in the recursion (ending on a white vertex)
- backward edges: edges ending in a grey vertex (including self loops)
- forward edges: edges (n, m) ending in a black vertex, and $d[n] < d[m]$
- cross edges: edges (n, m) ending in a black vertex, and $d[m] < d[n]$

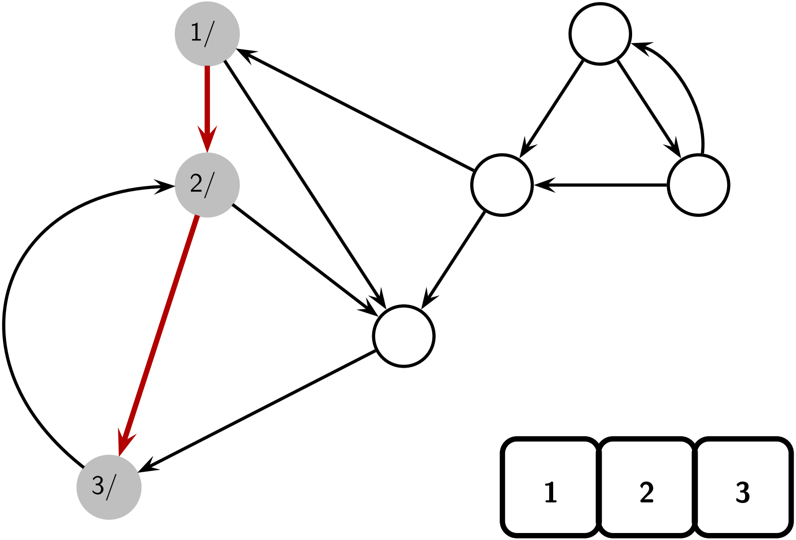
DFS example



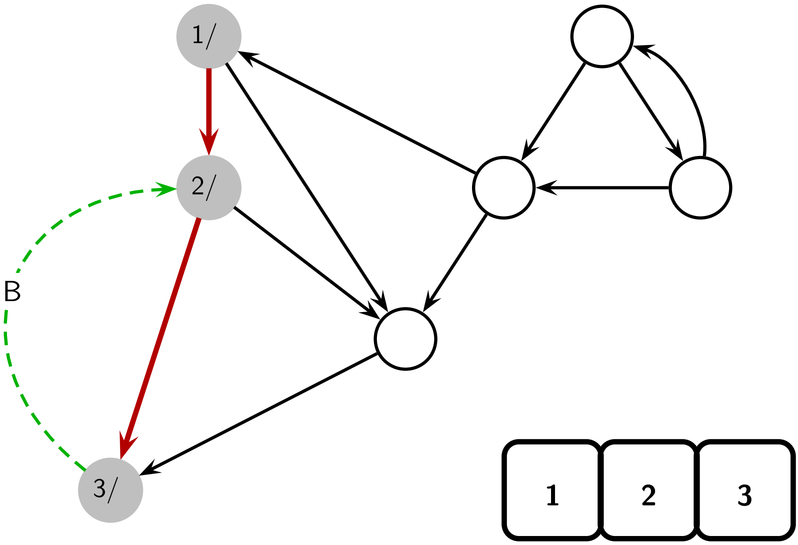
DFS example



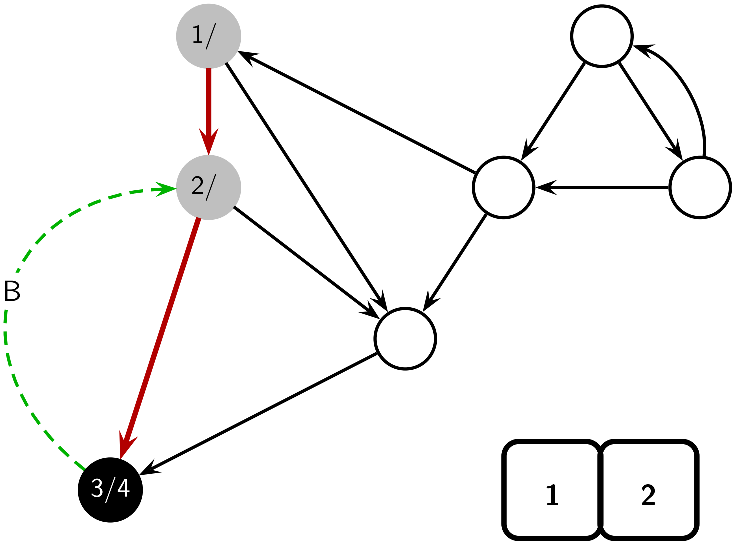
DFS example



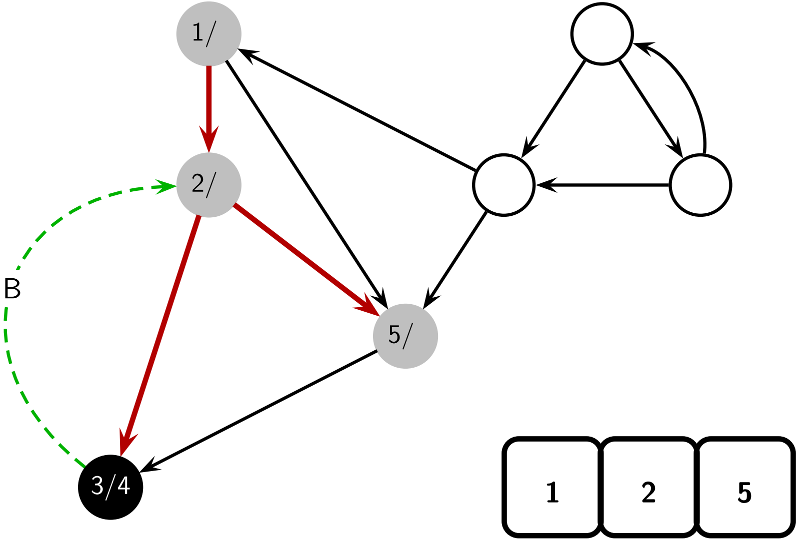
DFS example



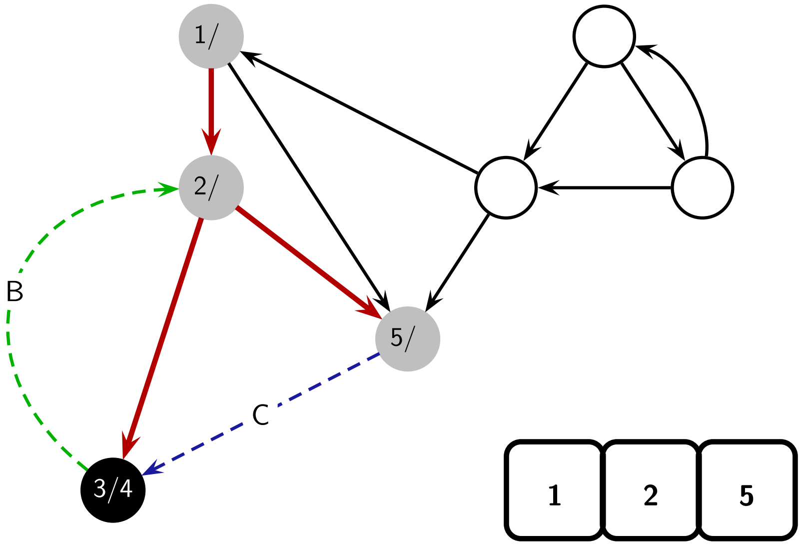
DFS example



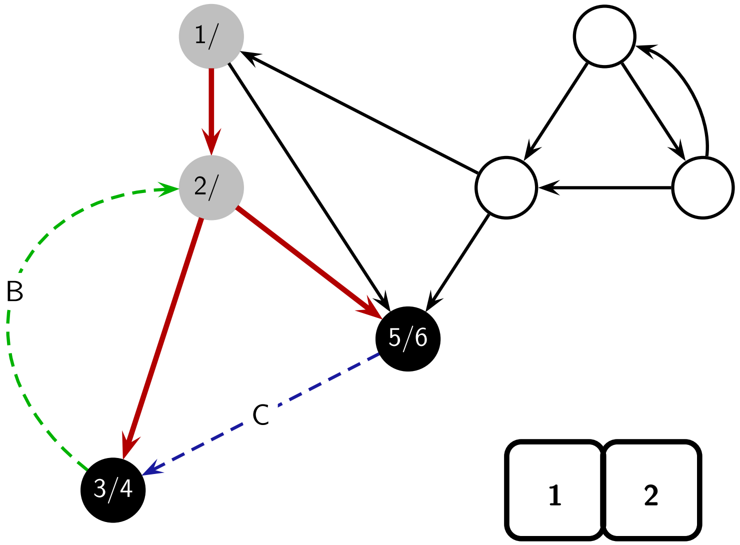
DFS example



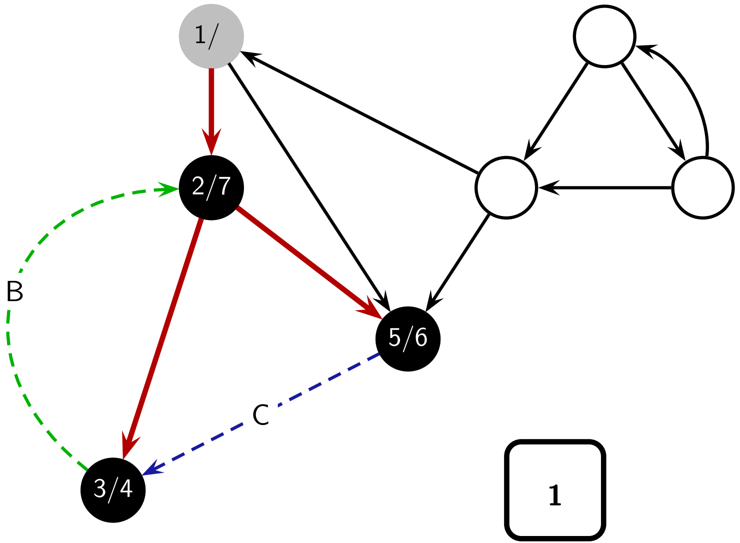
DFS example



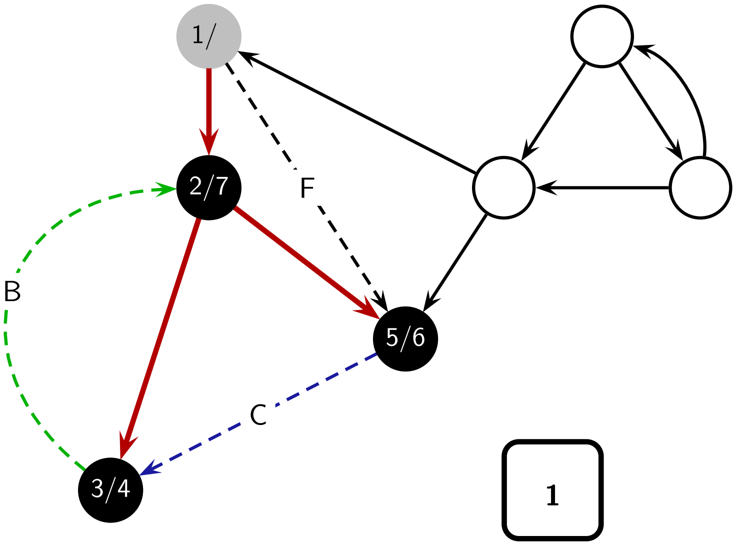
DFS example



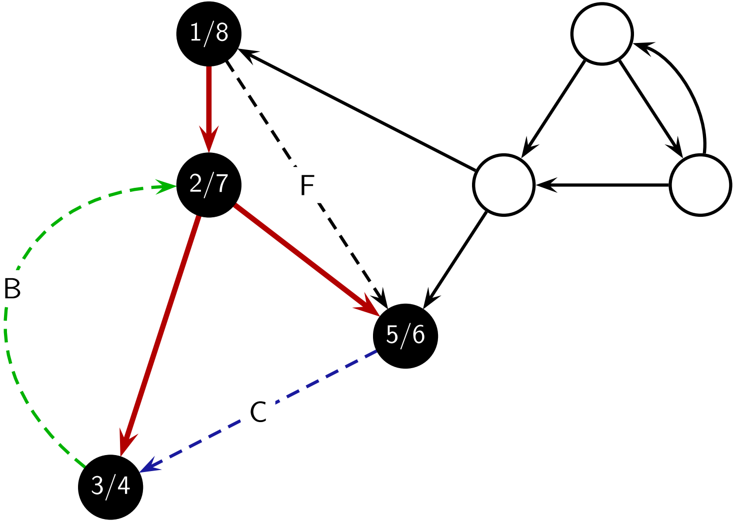
DFS example



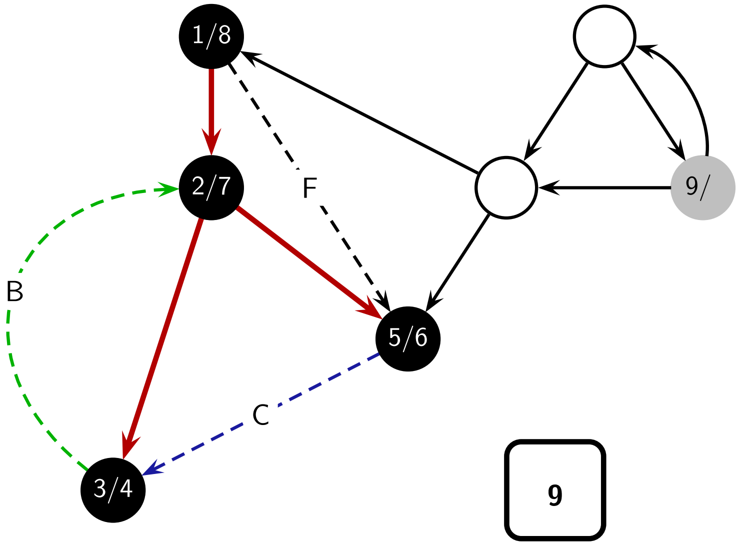
DFS example



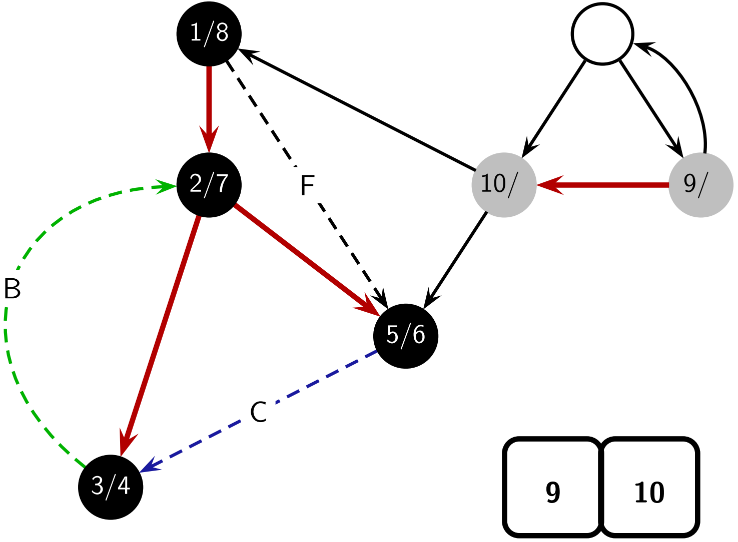
DFS example



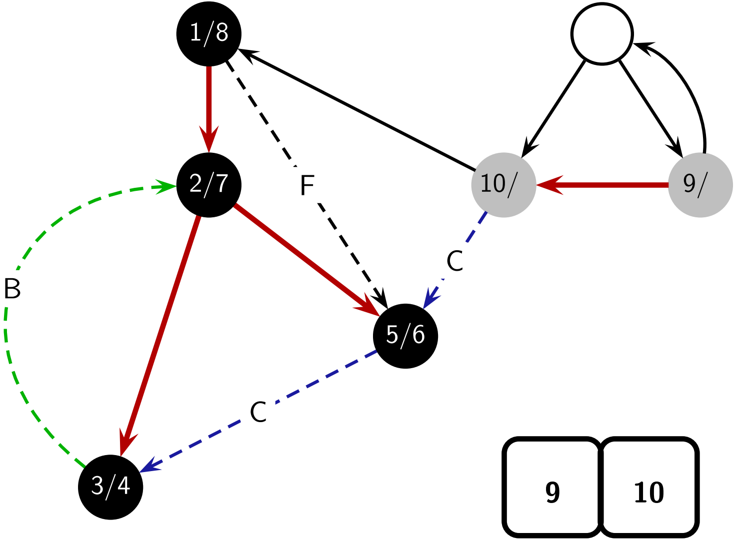
DFS example



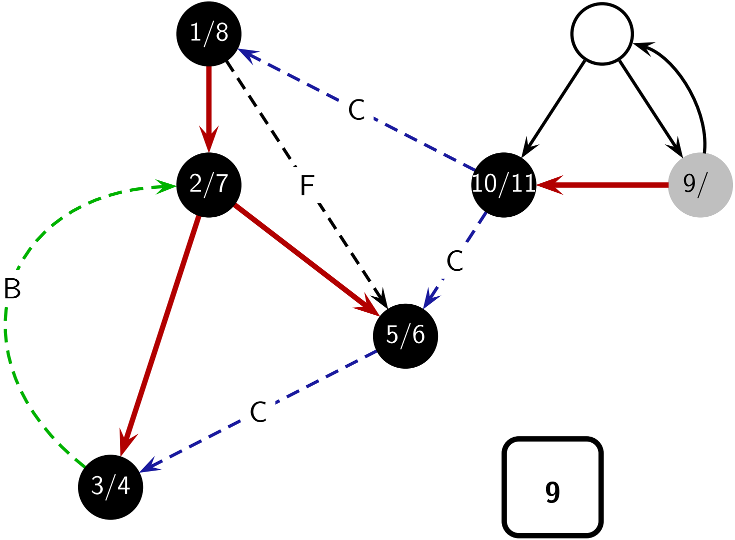
DFS example



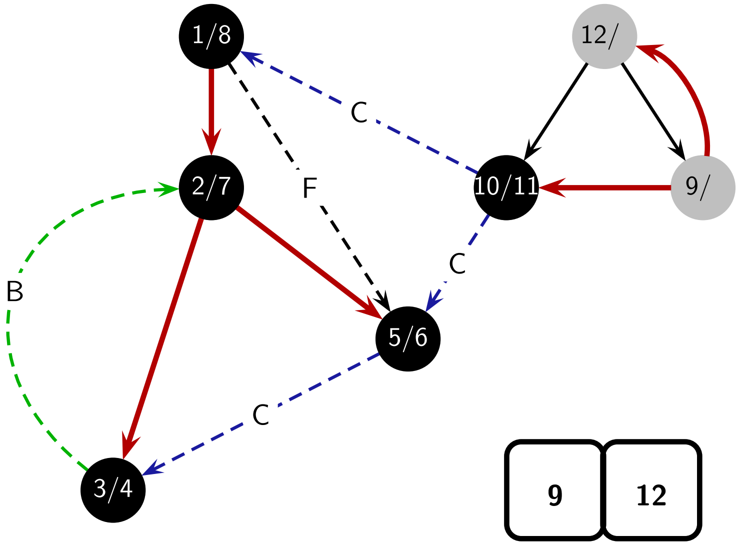
DFS example



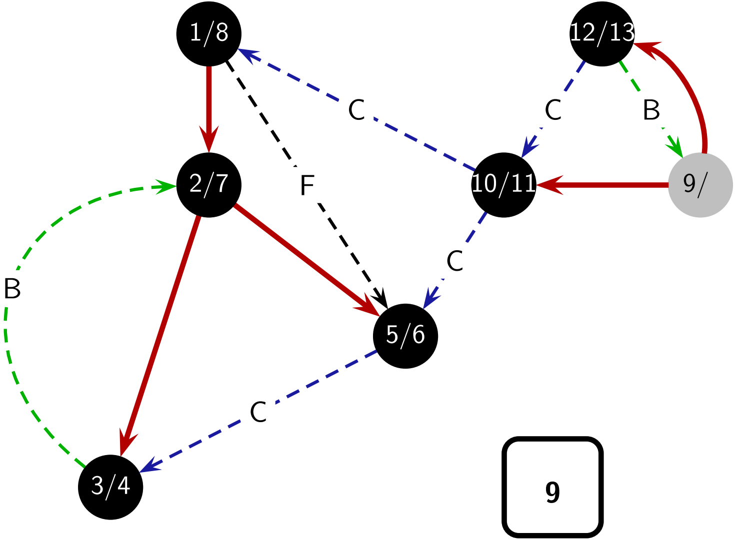
DFS example



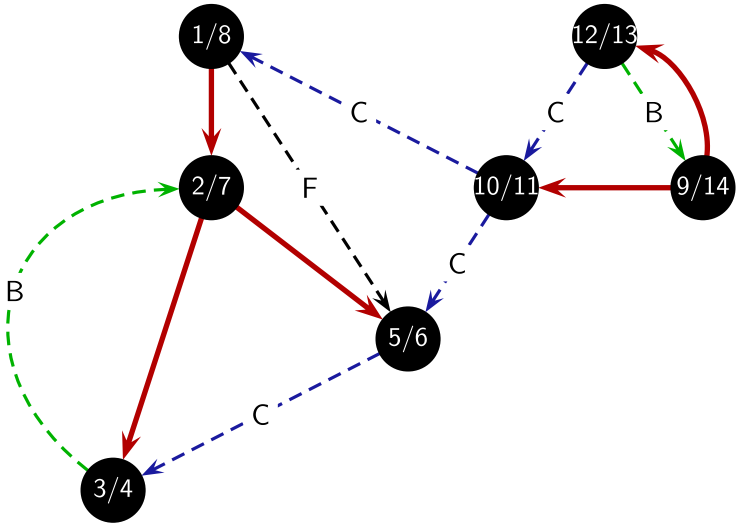
DFS example



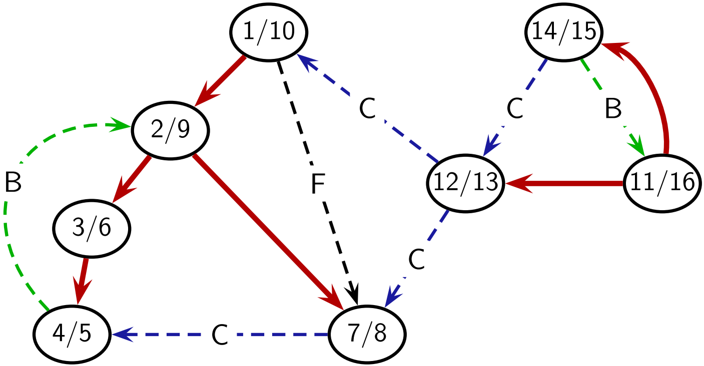
DFS example



DFS example



Edge Classification Example



During DFS, edge (s, t) is

Tree: $f(t) = 0 \wedge d(s) < d(t)$ Back: $f(t) = 0 \wedge d(s) > d(t)$

Cross: $f(t) \neq 0 \wedge d(t) < d(s)$ Forward: $f(t) \neq 0 \wedge d(t) > d(s)$

After DFS: $f(t) = 0 \Rightarrow f(t) > f(s)$ and $f(t) \neq 0 \Rightarrow f(t) < f(s)$

Note: a directed graph is acyclic if there are no back edges.

DFS/BFS Visitors

- Recap: Visitor Pattern, a Behavioural Pattern
- Purpose: Add functionality to a class without changing it
- Implementation:
  - Methods of class A get a visitor object as argument
  - The visitor's interface methods are called at specific points of the computation and have an A object as argument (at least)
  - This allows different additional computations or side effects with one class method of class A
  - The functionality is parameterized by the different visitor objects and classes, so to speak
- Especially useful with traversal methods of complex objects (like DFS or BFS)

DFS/BFS Visitors II

- Methods common for DFS/BFS visitor interface
  - `startNode(v,g)` : *v* is white in the outer loop
  - `discoverNode(v,g)` : *v* changes from white to gray
  - `finishNode(v,g)` : *v* changes from gray to black
  - `treeEdge(e,g)` : visit edge with white target node
- Methods specific to BFS visitor
  - `examineNode(v,g)` : *v* is taken off the queue
  - `grayTarget(e,g)` : gray target node
  - `blackTarget(e,g)` : black target node
- Methods specific to DFS visitor:
  `backEdge(e,g)`, `forwardEdge(e,g)`, `crossEdge(e,g)`

Topological Order

- Order the vertices such that if (n, m) is an edge, n comes before m
- Only exists for acyclic graphs
- Algorithm: sort vertices according to decreasing finishing times of DFS
- This can be easily implemented by a DFS visitor
- As each vertex is finished, add it to the front of a linked list
- The visitor contains this list as member variable
- After all vertices have been visited by DFS, the visitor holds the result
- Application example: A constraint graph that locally specifies which action must precede another

Topological Order Example