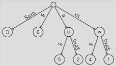


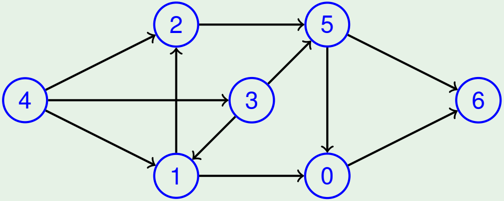
Datenstrukturen und Effiziente Algorithmen

Vorlesung *Datenstrukturen und Effiziente Algorithmen* im WS 18/19

Marc Hellmuth
Institut für Mathematik und Informatik
Universität Greifswald

Adjacency List Representation of a Graph

Example 1



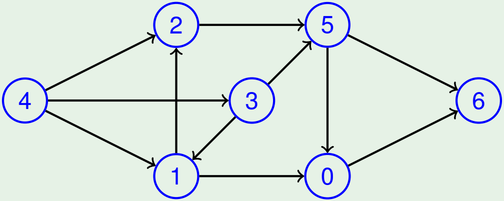
$V = \{0, 1, 2, 3, 4, 5, 6\}$
$E = \{(4, 2), (2, 5), (5, 6), (4, 3), (3, 5), (5, 0), (1, 2), (4, 1), (3, 1), (1, 0), (0, 6)\}$

Let $G = (V, E)$ be a graph with $n := |V|$ vertices. Let $V = \{0, 1, \ldots, n-1\}$. The adjacency list representation of G consists of

- an array of n adjacency lists, say $a[0], \ldots, a[n-1]$
- for vertex $u \in V$, $a[u]$ is the list of all $v \in V$ with an edge from u to v:
  - $(u, v) \in E$ (directed graphs)
  - $\{u, v\} \in E$ (undirected graphs)

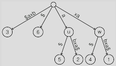
Adjacency List Representation of a Graph

Example 2



$a[0] = (6)$
$a[1] = (0, 2)$
$a[2] = (5)$
$a[3] = (5, 1)$
$a[4] = (2, 1, 3)$
$a[5] = (0, 6)$
$a[6] = ()$

- The space required to store the graph in this representation is $O(|V| + |E|)$.
- Vertex IDs do not need to be $0, 1, \ldots, n-1$. Options:
  1. Make IDs an attribute of the vertex, e.g. `v.id` in object-oriented programming
  2. Use another array with ids, e.g. $id[0], \ldots id[n-1]$
  3. Instead of an array use a data structure that allows other indexing: A hash allows to access a node like `a["Greifswald"]`



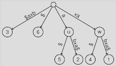
Adjacency List Representation of a Graph

C++ example **(just one of many ways to code a graph in adjacency list representation)**

```
class Node {
    string ID;
    list<Edge> adj;
};

class Edge {
    int weight;
    Node *from;
    Node *to;
};

class Graph {
    vector<Node> nodes;
};
```



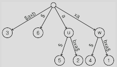
Alternative: Adjacency Matrix Representation of a Graph

Adjacency matrix

Edges are represented by a binary matrix $A = (a_{ij})_{0 \leq i,j < n}$

$$a_{ij} = \begin{cases} 1 & \text{, if } (i,j) \in E \\ 0 & \text{, otherwise.} \end{cases}$$

- requires $O(|V|^2)$ space which is often asymptotically larger than $O(|V| + |E|)$ (sparse/dense graphs)
- checking the presence of an edge takes only constant time (adjacency list: $O(|k|)$, where k is the length of the adjacency list)



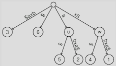
Breadth-First Search

BFS(G, s)

Let $G = (V, E)$ be a (directed) graph and $s \in V$ be the source.

$Q \leftarrow$ empty queue
for each vertex $v \in V \setminus \{s\}$ **do**
    $v.\pi \leftarrow NULL$ // predecessor during run of BFS
    $v.d \leftarrow \infty$ // distance to s
    v.color $\leftarrow$ white // white: not queued yet
$s.d \leftarrow 0$, s.color $\leftarrow$ gray, $s.\pi \leftarrow NULL$
enqueue(Q, s) // insert s into Q
while Q not empty **do**
    $u \leftarrow$ dequeue(Q, s) // $u =$ first element of Q
    for each $v \in Adj[u]$ **do**
        if v.color $=$ white **then**
            $v.d \leftarrow u.d + 1$
            $v.\pi \leftarrow u$
            v.color $\leftarrow$ gray
            enqueue(Q, v)
    v.color $\leftarrow$ black

Running time: $O(|V| + |E|)$



Breadth-First Search

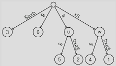
Properties (proof chalkboard)

Let $v \in V$ be any node. After running BFS(G, s)

1. $v.d$ is the distance $d(s, v)$ (length of a shortest path) from s to v.
2. A shortest path from s to v is obtained (in reverse order) by following the links $*.\pi$ starting from v and until reaching s.

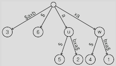
Remark

BFS can be considered a special case of Dijkstra's algorithm. The latter finds shortest paths in a *weighted* graph and uses a priority queque instead of an ordinary queque.



Depth-First Search (DFS, Tiefensuche)

- runs on directed and undirected graphs $G = (V, E)$
- is a graph traversal algorithm: it determines an ordering for the nodes, that
  - is every useful for many algorithms on trees that require bottom-up traversal
  - defines a so-called topological ordering on DAGs (more later)
- determines a forest on the node set V
  - each vertex $v \in V$ will receive a predecessor $v.pred \in V \cup \{\mathrm{NULL}\}$
  - the depth-first forest is (V, E_{pred}), where $E_{pred} := \{(v.pred, v) \mid v \in V, v.pred \neq \mathrm{NULL}\}$
  - if $v.pred = \mathrm{NULL}$ then v is a root in the forest otherwise $v.pred$ is v's father.
- we will assume an adjacency list representation for the time analysis



Depth-First Search

Node colors

white: The vertex has not yet been discovered.

gray: The vertex has been discovered but is not yet finished.

black: The vertex is finished: The vertex and all outgoing edges have been visited.

Depth-First Search

DFS(*G*)

```
for each vertex v of G do
    v.color ← white
    v.pred ← NULL
time ← 0
for each vertex v of G do
    if v.color = white then
        DFS-Visit(G, v)
```

DFS-VISIT(*G*, *v*)

```
time ← time + 1
v.discoverTime ← time
v.color ← gray
for each w in the adjacency list of v do
    if w.color = white then
        w.pred ← v
        DFS-VISIT(G, w)
time ← time + 1
v.finishTime ← time
v.color ← black
```

Depth-First Search

DFS(*G*)

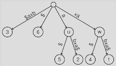
```
1: for each vertex v of G do
2:     v.color ← white
3:     v.pred ← NULL
4: time ← 0
5: for each vertex v of G do
6:     if v.color = white then
7:         DFS-Visit(G, v)
```

DFS-VISIT(*G*, *v*)

```
1: time ← time + 1
2: v.discoverTime ← time
3: v.color ← gray
4: for each w in the adjacency list of v do
5:     if w.color = white then
6:         w.pred ← v
7:         DFS-VISIT(G, w)
8: time ← time + 1
9: v.finishTime ← time
10: v.color ← black
```

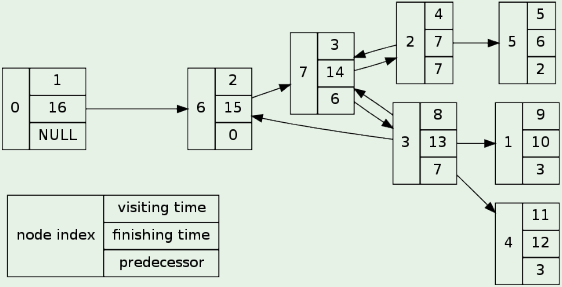
Running time

Loops 1-3 and 5-7 of DFS(G) each take time $O(|V|)$, not counting the time spent in DFS-VISIT(v). DFS-VISIT(v) is called *exactly once* for each vertex. Loop 3-6 of DFS-VISIT(v) is executed |a[v]| times. As $\sum_v |a[v]| = |E|$, we obtain the total running time $O(|V| + |E|)$.



Depth-First Search

Example 3



Classification of edges

Note that the depth-first forest is indeed a forest (exercise: prove that (V, E_{pred}) is acyclic). Edges $(u, v) \in E$ are either

- tree edges: $(u, v) \in E_{pred}$
- forward edges: not a tree edge and v is a proper descendant of u in the depth-first forest
- back edges: not a tree edge and v is an ancestor of u in the depth-first forest (includes self-loops)
- or cross edges: all other edges

Classification of edges

Note that the depth-first forest is indeed a forest (exercise: prove that (V, E_{pred}) is acyclic). Edges $(u, v) \in E$ are either

- tree edges: $(u, v) \in E_{pred}$
- forward edges: not a tree edge and v is a proper descendant of u in the depth-first forest
- back edges: not a tree edge and v is an ancestor of u in the depth-first forest (includes self-loops)
- or cross edges: all other edges

Observation

v is a descendant of u iff v is discovered during the time when u is gray iff DFS-VISIT(v) is called recursively during the execution of DFS-VISIT(u).

Classification of edges

Note that the depth-first forest is indeed a forest (exercise: prove that (V, E_{pred}) is acyclic). Edges $(u, v) \in E$ are either

- tree edges: $(u, v) \in E_{pred}$
- forward edges: not a tree edge and v is a proper descendant of u in the depth-first forest
- back edges: not a tree edge and v is an ancestor of u in the depth-first forest (includes self-loops)
- or cross edges: all other edges

Observation

v is a descendant of u iff v is discovered during the time when u is gray iff DFS-VISIT(v) is called recursively during the execution of DFS-VISIT(u).

Node colors partially determine edge class

If in the loop of line 3 of DFS-VISIT(v)

- w is white. Then (v, w) is a tree edge.
- w is gray. Then (v, w) is a back edge.
- w is black. Then (v, w) is a forward or cross edge.

Properties of DFS

Theorem 4 (Parenthesis theorem)

For a vertex v let $v.d$ be short for $v.discoverTime$ and $v.f$ be short for $v.finishTime$. Let u and v be two different vertices in G. After a run of DFS*(G) exactly one of the following three statements holds*

1. $[u.d, u.f] \cap [v.d, v.f] = \emptyset$ *and neither of the two vertices is a descendant of the other*
2. $[u.d, u.f] \subset [v.d, v.f]$ *and u is a descendant of v in a depth-first-tree*
3. $[u.d, u.f] \supset [v.d, v.f]$ *and u is an ancestor of v in a depth-first-tree.*

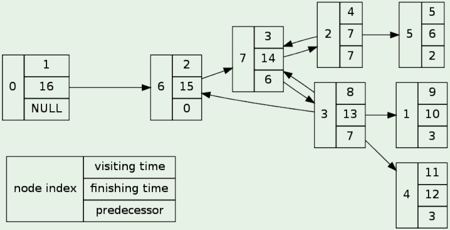
Properties of DFS

Theorem 4 (Parenthesis theorem)

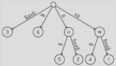
For a vertex v let $v.d$ be short for $v.discoverTime$ and $v.f$ be short for $v.finishTime$. Let u and v be two different vertices in G. After a run of DFS*(G) exactly one of the following three statements holds*

1. $[u.d, u.f] \cap [v.d, v.f] = \emptyset$ *and neither of the two vertices is a descendant of the other*
2. $[u.d, u.f] \subset [v.d, v.f]$ *and u is a descendant of v in a depth-first-tree*
3. $[u.d, u.f] \supset [v.d, v.f]$ *and u is an ancestor of v in a depth-first-tree.*

Example 5



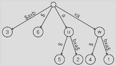
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
(	(	(	(	(	)	)	(	(	)	(	)	)	)	)	)
0	6	7	2	5	5	2	3	1	1	4	4	3	7	6	0



Properties of DFS

Proof.

Without loss of generality assume that u was discovered before v, i.e. $u.d < v.d$. Then a) $v.d < u.f$ or b) $v.d > u.f$.
In case a) v was discovered while u was gray. Therefore u is an ancestor of v. v must be finished before u, i.e. $v.f < u.f$. Therefore, case 3 of the theorem applies.
In case b) $u.d < u.f < v.d < v.f$ and case 1 of the theorem aplies. □

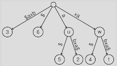


Properties of DFS

Theorem 6 (White path theorem)

Vertex v is a descendant of vertex u in the depth-first forest constructed by DFS*(G) if and only if at time $u.d$ there is a path in G from u to v consisting entirely of white vertices.*

(We consider a vertex to be white until right after it is discovered and each vertext is considered its own descendant.).



Properties of DFS

Theorem 6 (White path theorem)

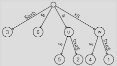
Vertex v is a descendant of vertex u in the depth-first forest constructed by $\text{DFS}(G)$ *if and only if at time $u.d$ there is a path in G from u to v consisting entirely of white vertices.*

(We consider a vertex to be white until right after it is discovered and each vertext is considered its own descendant.).

Proof.

$\Rightarrow$: Let v be a descendant of u in the depth-first forest. Then any vertex w on the path from u to v is also a descendant of u. Then case 3 of the parenthesis theorem holds and $[u.d, u.f] \supset [w.d, w.f]$, which implies $u.d < w.d$. As w is discovered after u, vertex w is white at time $u.d$.

$\Leftarrow$: Suppose at time $u.d$ there is a path π from u to v consisting entirely of white vertices. Assume, for the sake of contradiction, that v is not a descendant of u. Then, there are vertices r and s on π such that $(r, s) \in E$, r is a descentant of u, but s is not a descendant of u ($r = u$ is possible). By the parenthesis theorem, $u.d < r.d < r.f < u.f$. As s is not a descendant of u it must remain white during the time interval $[u.d, u.f]$. As there is an edge from r to s, when the loop in line 3 is executed during the call to DFS-VISIT(r) vertex s is discovered: $r.d < s.d < r.f$. By the parenthesis theorem s must also be a descendant of u, which consitutes the desired contradiction. □



Topological Ordering

Definition 7 (Topologial ordering)

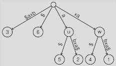
A topological ordering of a directed graph $G = (V, E)$ with n vertices is an ordering $s = (v_1, \dots, v_n)$ of the vertices V (i.e. $V = \{v_1, \dots, v_n\}$) such that

$$i < j \text{ for all } (v_i, v_j) \in E.$$

Example 8

dressing

(chalk board)



Topological Ordering

Definition 7 (Topologial ordering)

A topological ordering of a directed graph $G = (V, E)$ with n vertices is an ordering $s = (v_1, \ldots, v_n)$ of the vertices V (i.e. $V = \{v_1, \ldots, v_n\}$) such that

$$i < j \text{ for all } (v_i, v_j) \in E.$$

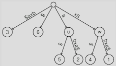
Example 8

dressing

(chalk board)

DAG

When G contains a cycle, then no topologial ordering can exist. We will below give an algorithm that constructs a topological ordering for any DAG, however.

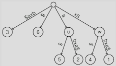


Topological Ordering

TOPOLOGICAL-SORT(G)

1. initialize s as the empty list
2. call a variant of DFS(G), where DFS-VISIT(v) has an additional line:
   9: insert v at the front of s
3. return s

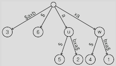
> The topological order is the reverse order of finishing times.



Topological Ordering

Theorem 9

A directed graph G has a cycle iff $\mathsf{DFS}(G)$ *yields at least one back edge.*



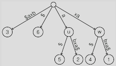
Topological Ordering

Theorem 9

A directed graph G has a cycle iff $\mathrm{DFS}(G)$ *yields at least one back edge.*

Proof.

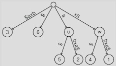
$\Leftarrow$: Suppose (u, v) is a back edge. Then v is an ancestor of u in the depth-first forest produced by $\mathrm{DFS}(G)$. Therefore, there is a path in G from v to u which becomes a cycle by adding the edge (u, v) to it.
$\Rightarrow$: Suppose G has a cycle c. Let v be the vertex on the cycle that is discovered fist during $\mathrm{DFS}(G)$. Let u be the vertex preceeding v on the cycle c ($u = v$ is possible). By the white path theorem, and as all vertices on c are white at time $v.d$, u is a descendant of v in the depth-first forest. The edge (u, v) is not a tree edge, as the tree would otherwise contain cycle c. The edge (u, v) must therefore be a back edge. □



Topological Ordering

Theorem 10

For a DAG G, TOPOLOGICAL-SORT*(G) returns a topological ordering of the vertices of G.*



Topological Ordering

Theorem 10

For a DAG G, TOPOLOGICAL-SORT*(G) returns a topological ordering of the vertices of G.*

Proof.

Let G be acyclic and let $(v, w) \in E$ be any edge. Consider the point in time when this edge is explored (line 3 of DFS-VISIT(v)). If w is white at that time, then DFS-VISIT(w) is called and w is finished before v: $w.f < v.f$. w cannot be gray at that time, as otherwise (v, w) would be a back edge and by Theorem 9 G would not be acyclic. If w is black at that time, then it has already been finished and also $w.f < v.f$. In any case all edges go from a vertex with a later finishing time to a vertex with an earlier finishing time. Therefore, the reversed finishing times consitute a topological ordering. □