# Algorithms for Data Processing
# Lecture III: Graph Algorithms – Directed Graphs

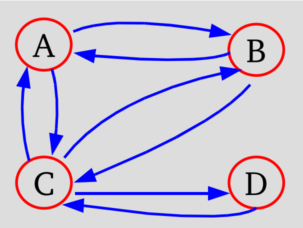
**Alessandro Artale**

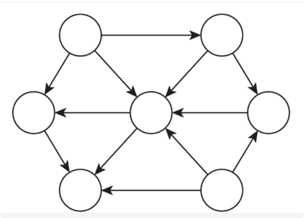
Free University of Bozen-Bolzano
Faculty of Computer Science
http://www.inf.unibz.it/~artale
artale@inf.unibz.it

2018/19 – First Semester
MSc in Computational Data Science — UNIBZ

# Directed Graphs

- The general definition of directed graph is similar to the definition of graph, except that one associates an ordered pair of vertices with each edge.
- Thus each edge of a directed graph can be drawn as an arrow going from the first vertex to the second vertex of the ordered pair.



# Representing Directed Graphs: Adjacency List

Each vertex has two lists associated with it:

- Direct List consists of vertices to which it has edges, and
- Reverse List ($G^{rev}$) consists of vertices from which it has edges.

# Search in Directed Graphs

Directed Reachability. Given a node s, find all nodes reachable from s.

Directed s-t shortest path problem. Given two nodes s and t, what is the length of a shortest directed path from s to t?

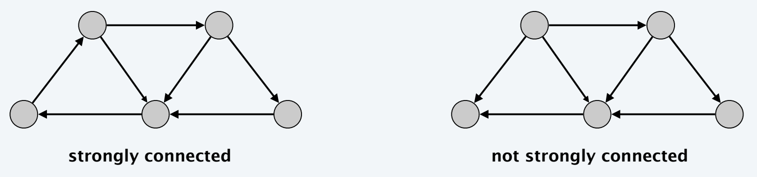
Graph search algorithms. BFS/DFS extend naturally to directed graphs to compute Directed reachability and Directed s-t shortest path using the Direct List.

Computing $T^{rev}$. Given a vertex s, and the Reverse List, $G^{rev}$, BFS and DFS compute the set of nodes with paths pointing to s.

# Strong Connectivity

Def. Vertices u and v are mutually reachable if there is both a path from u to v and also a path from v to u.

Def. A graph $G$ is strongly connected if every pair of nodes is mutually reachable.



# Strong Connectivity/2

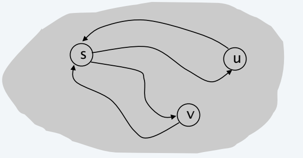
Property. Let s be any vertex. $G$ is strongly connected iff

- every vertex is reachable from s, and
- s is reachable from every vertex.

**Proof.**

($\Rightarrow$) By definition.

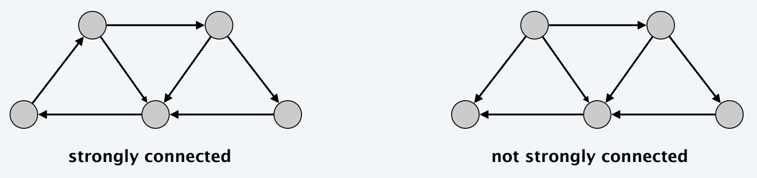
($\Leftarrow$) Path from $u$ to $v$: concatenate $u \rightarrow s$ path with $s \rightarrow v$ path.
Path from $v$ to $u$: concatenate $v \rightarrow s$ path with $s \rightarrow u$ path.



# Strong Connectivity: Algorithm

- Pick any node $s$.
- Run BFS from s using the Direct List to compute set `Discovered`.
- Run BFS from s using the Reverse List to compute set `Discovered-Rev`.
- Return true iff $|\texttt{Discovered}| = |\texttt{Discovered-Rev}| = |V|$.

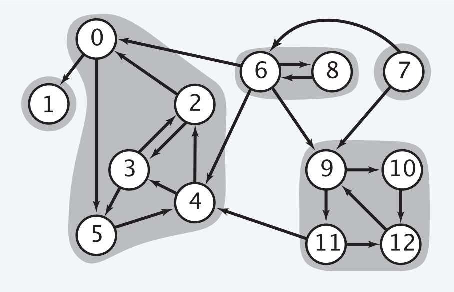
Complexity of Strong Connectivity: $O(m + n)$.



## Strong Components

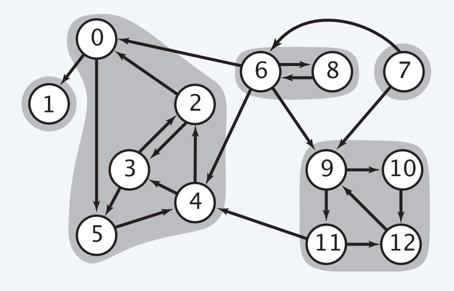
By analogy with connected components in an undirected graph, we can define:

- Strong component of a graph as a maximal subset of mutually reachable vertices.



# Strong Components/2

Property. For any two nodes $s$ and $t$ in a directed graph, their strong components are either identical or disjoint.



# Strong Components – Tarjan's Algorithm

- Tarjan's algorithm (1972, from the name of it's inventor) is based on depth first search (DFS).
- The vertices are increasingly indexed as they are traversed by the DFS procedure.
- While returning from the recursion of DFS, every vertex $v$ gets assigned a vertex $v_R$ as a representative.
- $v_R$ is a vertex with the least index that can be reached from $v$.
- Vertices with the same representative are located in the same strongly connected component.

# Strong Components – Tarjan's Algorithm/2

```
Strong-Components(G)
index=1; v.index=0, for all v ∈ V;        /* index set to 1 and vertex index set to 0 */
S = [];                                   /* The stack is initialized empty */
for each v ∈ V                            /* depth-first search for each vertex */ do
    if v.index=0 then Tarjan(v)           /* that was not already visited */

Tarjan(v);
v.index = index; v.min-index = index;     /* set vertex indices to the current index */
index = index + 1; S.push(v);
for each edge (v, v') incident to v       /* checks all vertices adjacent to v */ do
    if v'.index=0 then                    /* vertex not already visited */
        Tarjan(v');                       /* DFS Recursion */
        v.min-index = min(v.min-index, v'.min-index)
    else if v' is inside S then
        v.min-index = min(v.min-index, v'.min-index);   /* v' has a path to v */
if v.min-index = v.index;   /* v is a representative vertex and an SCC has been found */
 then
    repeat
        v' = S.pop();
        output v';                        /* output SCC */
    until (v' != v);
```

# Tarjan Complexity

- The `For-Loop` takes $O(\mathbf{deg}(v))$ time for each vertex $v$;
- Thus, in total we need $O(\Sigma_{v \in V} \mathbf{deg}(v))$;
- From graph properties, $\Sigma_{v \in V} \mathbf{deg}(v) = 2m$;
- Thus, $O(\Sigma_{v \in V} \mathbf{deg}(v)) = O(m)$;
- We need $O(n)$ additional time to check whether a vertex has been already visited and to keep the information on whether a vertex is currently inside the stack (e.g., using a boolean array `IsInsideStack`$[n]$);
- Finally, Tarjan runs in $O(m + n)$.

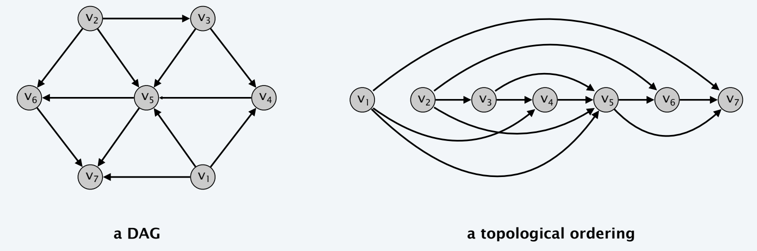
# Directed acyclic graphs

A DAG is a directed graph that contains no directed cycles

- DAGs can be used to encode *precedence relations* or *dependencies*.
  - Suppose we have a set of tasks labeled $T_1$, $T_2 \dots T_n$ that need to be performed, and an edge $(i, j)$ means that $T_i$ must be performed before $T_j$ (*Job Scheduling*).
- Given a set of tasks with dependencies, it would be natural to seek a valid order in which the tasks could be performed, so that all dependencies are respected.

# Topological Order

A topological order of a directed graph $G = (V, E)$ is an ordering of its nodes as $v_1, v_2, \ldots, v_n$ so that for every edge $(v_i, v_j)$ we have $i < j$.

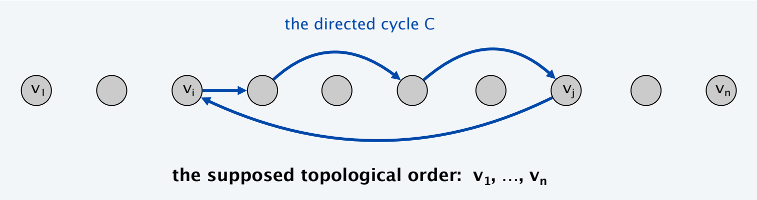


# Directed Acyclic Graphs and Topological Order

Property/1. If a directed $G$ has a topological order, then $G$ is a DAG.

Proof by Contradiction.

- Suppose that $G$ has a topological order $v_1, v_2, \dots, v_n$ and that $G$ also has a directed cycle $C$.
- Let $v_i$ be the lowest-indexed node in $C$, and let $v_j$ be the node just before $v_i$; thus $(v_j, v_i)$ is an edge, and $i < j$.
- On the other hand, since $(v_j, v_i)$ is an edge and $v_1, v_2, \dots, v_n$ is a topological order, we must have $j < i$, a contradiction.



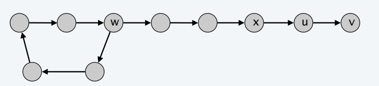
# Directed Acyclic Graphs and Topological Order/2

Also the vice-versa holds: If $G$ is a DAG then $G$ has a topological order. We first show the following:

Property/2. If $G$ is a DAG, then $G$ has a node with no entering edges.

Proof by Contradiction.

- Let $G$ be a DAG and every node has at least one entering edge.
- Pick a node $v$, and follow an edge backward from $v$. Since v has at least one entering edge $(u, v)$ we can walk backward to $u$.
- Since $u$ has at least one entering edge $(x, u)$, we can walk backward to $x$.
- Repeat until we visit a node, say $w$, twice.
- Let $C$ denote the sequence of nodes encountered between successive visits to $w$. $C$ is a cycle.



# Directed Acyclic Graphs and Topological Order/3

Property/3. If $G$ is a DAG, then $G$ has a topological ordering.

Proof by Induction on the number of vertices.

- Base case: true if $n = 1$.
- Given DAG with $n > 1$ nodes, find a node $v$ with no entering edges.
- $G \setminus \{v\}$ is a DAG, since deleting $v$ cannot create cycles.
- By inductive hypothesis, $G \setminus \{v\}$ has a topological ordering, say $TO_{G\setminus\{v\}}$.
- Let $TO_G = v, TO_{G\setminus\{v\}}$, this is valid since $v$ has no entering edges.

# Topological Order – Algorithm in $O(n^2)$

The following Algorithm applies Property/3 to a DAG to obtain its topological order.

- To find a node with no incoming edges the Reverse List, $G^{rev}$, is used. This search costs $O(n)$.
- Repeting this step $n$–times we obtain a final cost of $O(n^2)$.

i=1; `/* Initialize i and call Topological-Order(G) */`

`Topological-Order(G)`;

Find a node $v$ in $G$ with no incoming edges;

**if** *$v$ does not exists* **then**

  **return**

**else**

  TO[i] = v;

  i=i+1;

  `Topological-Order(`$G \setminus \{v\}$`)`

# Topological Order – Algorithm in $O(m+n)$

To achieve a running time of $O(m+n)$ we do the following:

- Maintain the following information with an initialization cost of $O(m+n)$:
  - $\texttt{count}[w]$: Array counting the number of incoming edges to each node $w$;
  - $S$: set of nodes with no incoming edges (can be implemented as a stack or a queue);
- Before each recursive call (the following costs $O(\mathbf{deg}(v))$ time for each vertex $v$):
  - remove $v$ from $S$ (e.g., pop $v$ from $S$);
  - decrement $\texttt{count}[w]$ for all edges from $v$ to $w$, and add $w$ to $S$ if $\texttt{count}[w] = 0$;

**Exercise:** Pseudo-code for the Algorithm.

# Thank You!