Algorithms for Programming Contests - Week 03

Prof. Dr. Javier Esparza,
Vincent Fischer,
Jakob Schulz,
conpra@model.cit.tum.de

October 28, 2025

Graphs

Graphs

A *graph* is a tuple $G = (V, E)$, where V is a non-empty set of *vertices* and E is a set of edges.

A *directed* graph is a graph with $E \subseteq V \times V = \{(u, v) \mid u, v \in V\}$.

An *undirected* graphs is a graph with $E \subseteq \{\{u, v\} \mid u, v \in V\} =: \binom{V}{2}$.

For a vertex v, we denote the successors of v by
$vE := \{u \mid (v, u) \in E\}$ for directed graphs;
$vE := \{u \mid \{v, u\} \in E\}$ for undirected graphs.

- A *path* from v_1 to v_n is a sequence $p = v_1 v_2 \dots v_n$ such that $v_{i+1} \in v_i E$ for all $i \in [1, n-1]$. The path is called *simple* if $v_i \neq v_j$ for all $i \neq j$. *Note:* ε is a path from v_1 to v_1.
- A *cycle* is a simple path $v_1 v_2 \dots v_n$ s.t. $v_1 \in v_n E$ and, only in the undirected case, $n \geq 3$.
- A graph is *cyclic* if there exists a cycle, otherwise it is *acyclic*.

- A *path* from v_1 to v_n is a sequence $p = v_1 v_2 \dots v_n$ such that $v_{i+1} \in v_i E$ for all $i \in [1, n-1]$. The path is called *simple* if $v_i \neq v_j$ for all $i \neq j$. *Note:* ε is a path from v_1 to v_1.
- A *cycle* is a simple path $v_1 v_2 \dots v_n$ s.t. $v_1 \in v_n E$ and, only in the undirected case, $n \geq 3$.
- A graph is *cyclic* if there exists a cycle, otherwise it is *acyclic*.

- An undirected graph is *connected* if for every pair of vertices $u, v \in V$, there is a path from u to v.
- For an undirected graph, a *connected component* is a maximal (w.r.t. set inclusion) set $V' \subseteq V$ s.t. $(V', E \cap \binom{V'}{2})$ is connected.
- An undirected graph is a *tree* if it is acyclic and connected. For any tree (V, E), we have $|V| = |E| + 1$.
- An undirected acyclic graph is called a *forest*. Note that each connected component of a forest is a tree.

- A *path* from v_1 to v_n is a sequence $p = v_1 v_2 \dots v_n$ such that $v_{i+1} \in v_i E$ for all $i \in [1, n-1]$. The path is called *simple* if $v_i \neq v_j$ for all $i \neq j$. *Note:* ε is a path from v_1 to v_1.
- A *cycle* is a simple path $v_1 v_2 \dots v_n$ s.t. $v_1 \in v_n E$ and, only in the undirected case, $n \geq 3$.
- A graph is *cyclic* if there exists a cycle, otherwise it is *acyclic*.

- A directed graph is *strongly connected* if for every pair of vertices $u, v \in V$, there is a path from u to v.
- A directed graph is *connected* if $(V, E \cup E^T)$ is strongly connected, where $E^T := \{(v, u) \mid (u, v) \in E\}$.
- For a directed graph, a *strongly connected component* (SCC) is a maximal set $V' \subseteq V$ s.t. $(V', E \cap (V' \times V'))$ is strongly connected.
- A directed acyclic graph is also called a *DAG*.
- Note: For any directed graph, merging each SCC into a single node results in a DAG. (Exercise: prove!)

Graphs: basic interface

Graph operations

- Make graph: build a graph from a list of vertices and edges.
- Get vertices: Iterate over all vertices $v \in V$.
- Get edges: Iterate over all edges $e \in E$.
- Test edge: Test existence of an edge $(u, v) \in E$.
- Get successors: For a vertex v, iterate over all successors $u \in vE$.

Graphs: implementation

Graph representation

- Adjacency list: For each vertex v, store a list of successors vE.
- Adjacency matrix: For each pair of vertices u, v, store existence of an edge $(u, v) \in E$.

Which one to choose?

- Usually, adjacency lists are better.
- Adjacency matrices may be preferred if:
  - the graph is dense, i.e. $|E|$ is close to $|V|^2$,
  - edge testing is used extensively or
  - performance difference is irrelevant.

Graphs: implementation

Edge test/iteration: Adjacency matrix

```
struct Graph {
        n: usize,
        adj_matrix: Vec<Vec<bool>>,
}

impl Graph {
        fn has_edge(&self, u: usize, v: usize) -> bool {
                return self.adj_matrix[u][v];
        }
        fn iter_edges(&self) -> Vec<(usize, usize)> {
                let mut result = vec![];
                for u in 0..self.n {
                        for v in 0..self.n {
                                if self.adj_matrix[u][v] {
                                        result.push((u, v));
                                }
                        }
                }
                return result;
        }
}
```

Graphs: implementation

Edge test/iteration: Adjacency list

```
struct Graph {
        n: usize,
        adj_list: Vec<Vec<usize>>,
}

impl Graph {
        fn has_edge(&self, u: usize, v: usize) -> bool {
                return self.adj_list[u].contains(&v);
        }
        fn iter_edges(&self) -> Vec<(usize, usize)> {
                let mut result = vec![];
                for u in 0..self.n {
                        for &v in self.adj_list[u].iter() {
                                result.push((u, v));
                        }
                }
                return result;
        }
}
```

Graphs: implementation

Edge test/iteration: Adjacency list

```
struct Graph {
        n: usize,
        adj_list: Vec<Vec<usize>>,
}

impl Graph {
        fn has_edge(&self, u: usize, v: usize) -> bool {
                return self.adj_list[u].contains(&v);
        }
        fn iter_edges(&self) -> Vec<(usize, usize)> {
                let mut result = vec![];
                for u in 0..self.n {
                        for &v in self.adj_list[u].iter() {
                                result.push((u, v));
                        }
                }
                return result;
        }
}
```

can be made faster by using binary search or hash table!

Graph traversal

Graph traversal

Graph traversal

- Visit vertices in certain order.
- Assign vertices an order $o : V \to \mathbb{N} \cup \{\infty\}$ of discovery time.
- Possibly keep track of other information such as finishing time, predecessor, etc.

Usages

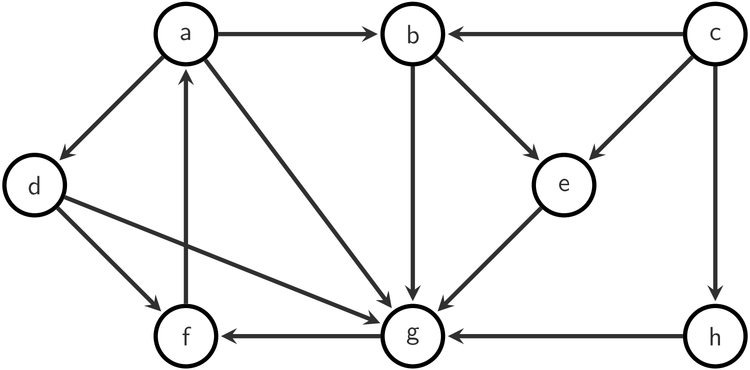
- Find vertex with certain properties.
- Check property for all vertices.
- Find (strongly) connected components.
- Check for cycles.
- ...

Depth First Search (DFS)

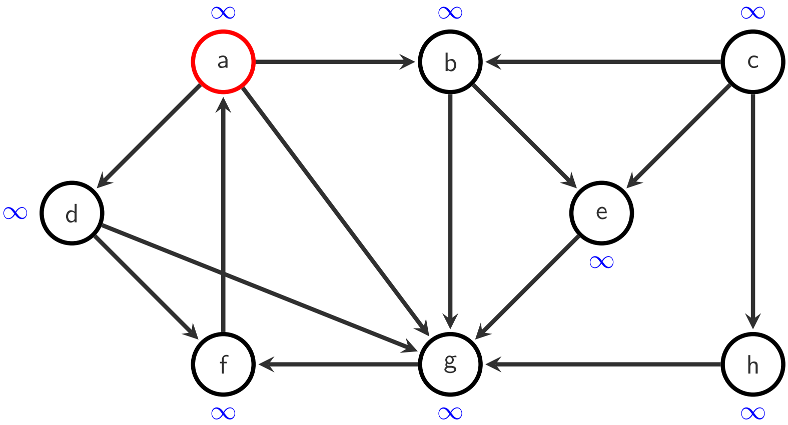
Basic Idea:

- Manage discovered vertices that still need to be explored in a stack.
- Repeat: pop element from stack and push its successors to stack.

Depth First Search (DFS)

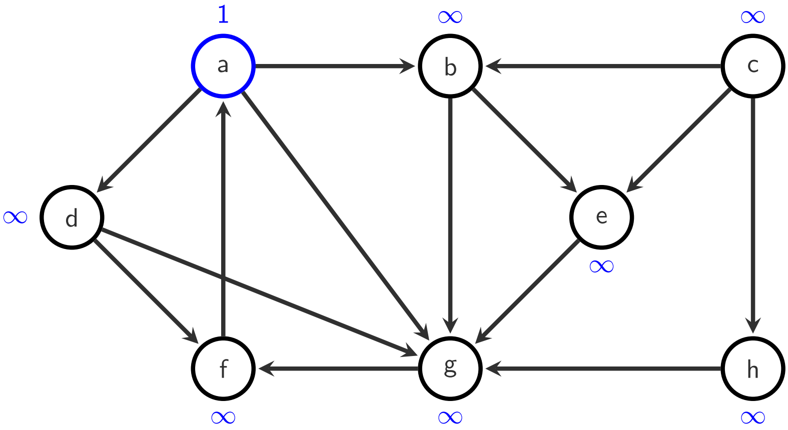


Depth First Search (DFS)



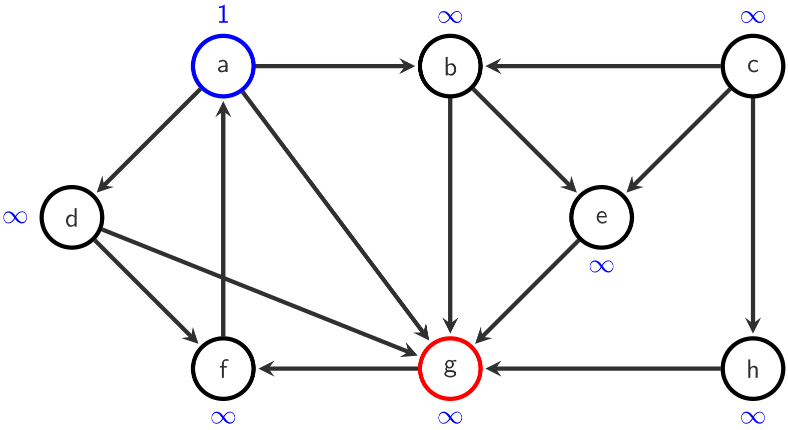
$S = [a]$

Depth First Search (DFS)



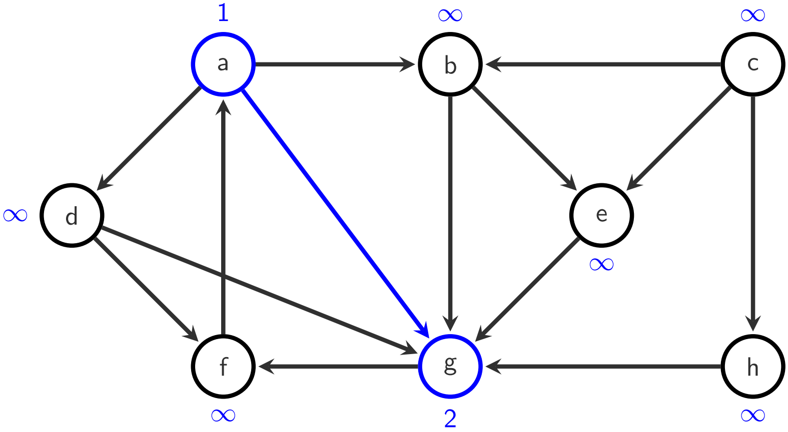
$S = [\mathrm{b}, \mathrm{d}, \mathrm{g}]$

Depth First Search (DFS)



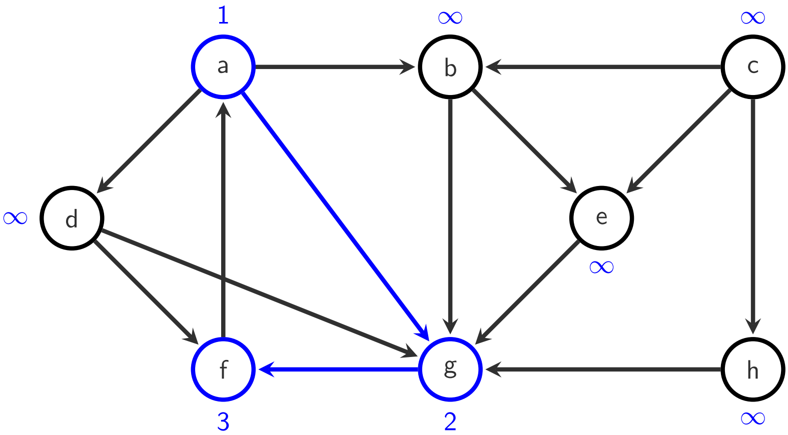
$S = [\text{b}, \text{d}, \textcolor{red}{\text{g}}]$

Depth First Search (DFS)



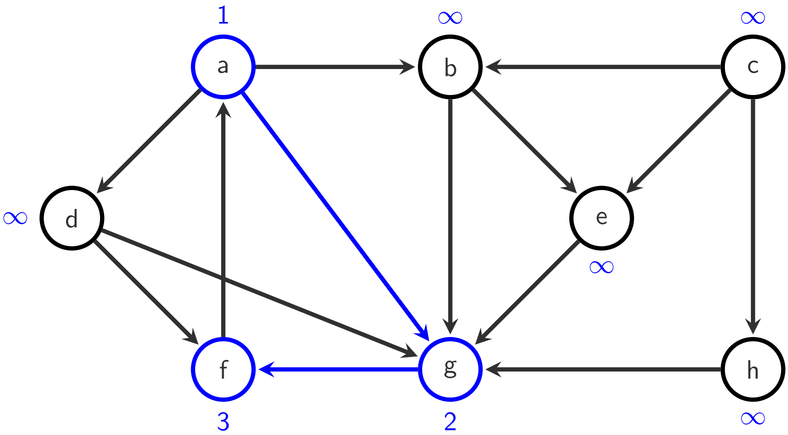
$S = [\text{b}, \text{d}, \text{f}]$

Depth First Search (DFS)



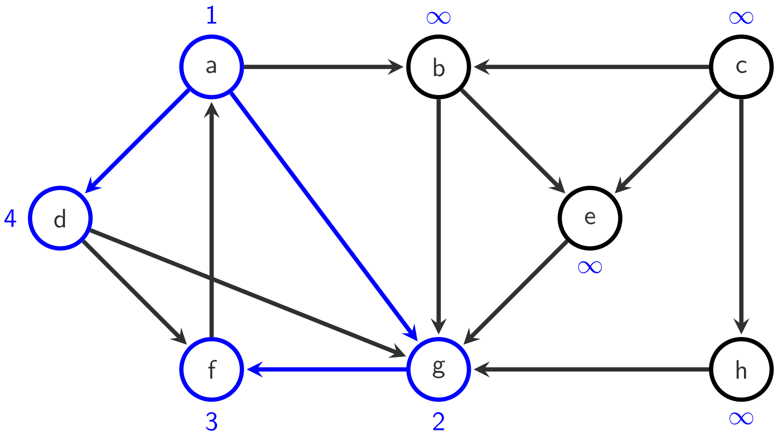
$S = [\mathrm{b}, \mathrm{d}, \mathrm{a}]$

Depth First Search (DFS)



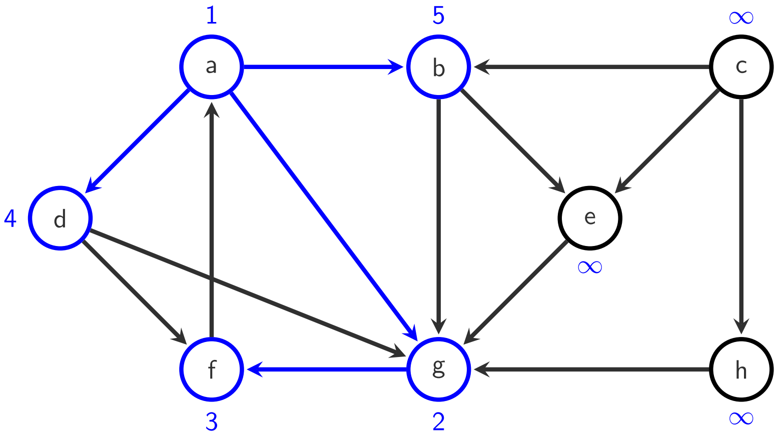
$S = [\mathrm{b}, \mathrm{d}]$

Depth First Search (DFS)



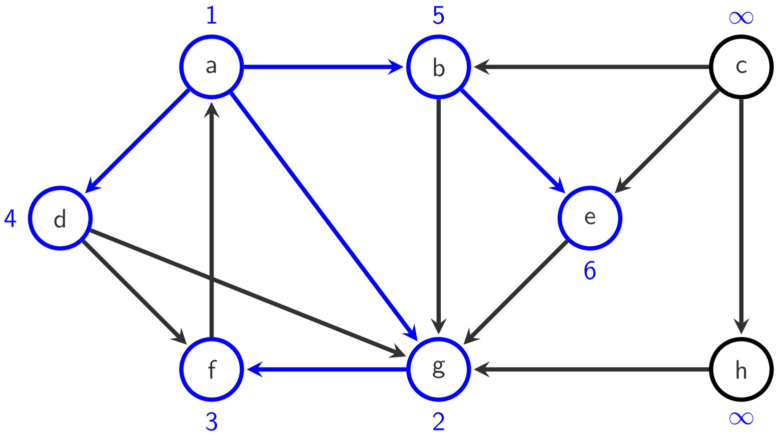
$S = [\mathrm{b}, \mathrm{f}, \mathrm{g}]$

Depth First Search (DFS)



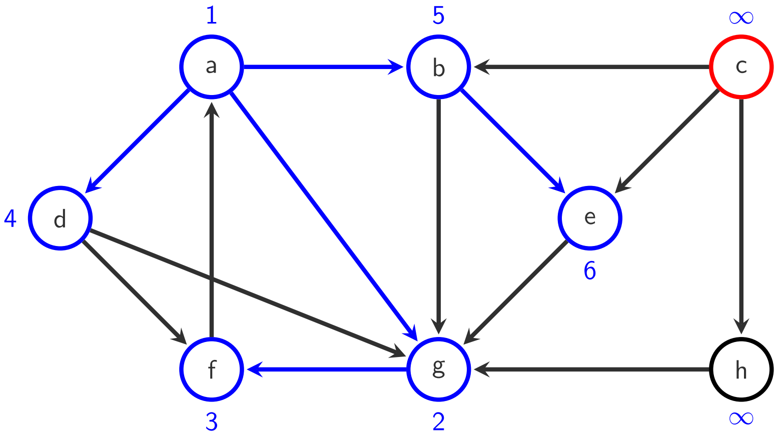
$S = [\text{e}, \text{g}]$

Depth First Search (DFS)



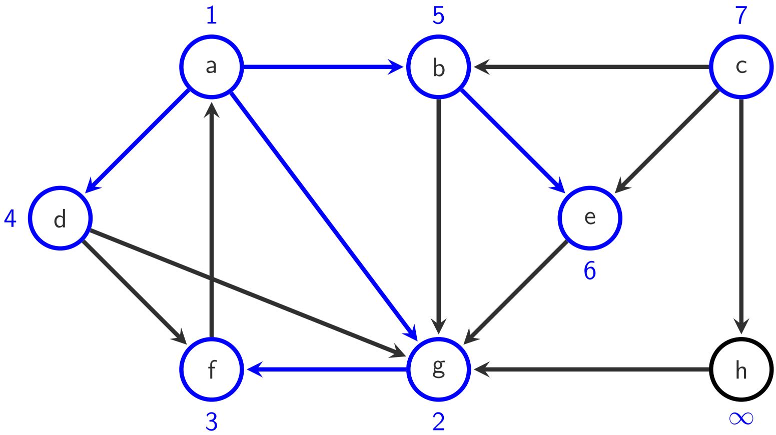
$S = [\mathrm{g}]$

Depth First Search (DFS)



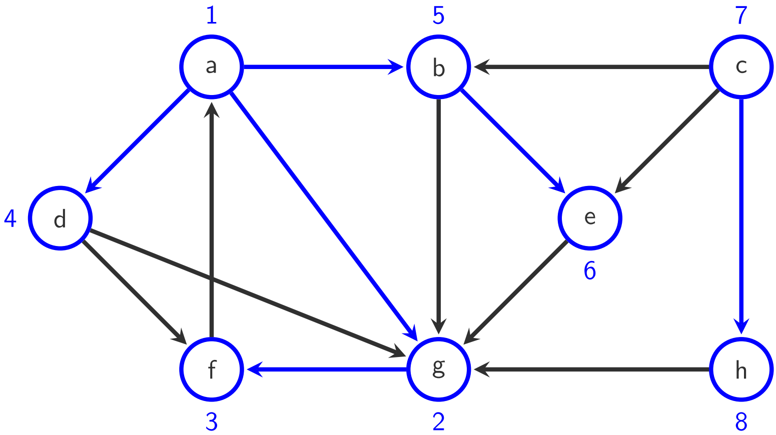
$S = [c]$

Depth First Search (DFS)



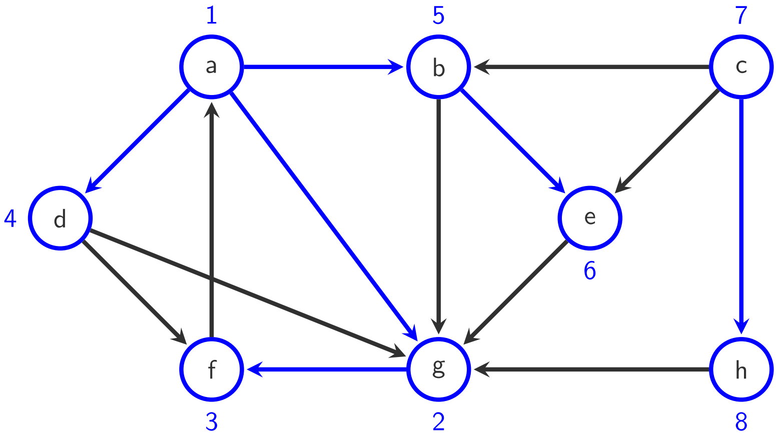
$S = [\text{e}, \text{h}]$

Depth First Search (DFS)



$S = [\mathrm{e}, \mathrm{g}]$

Depth First Search (DFS)



$S = []$

DFS Algorithm

Algorithm 1 Depth First Search

Input: Graph $G = (V, E)$

procedure $\textsc{DFS}(G)$
- **for** each vertex $v \in V$ **do**
  - $o(v) \leftarrow \infty$
- $S \leftarrow$ EmptyStack()
- $i \leftarrow 1$
- **for** each vertex $v \in V$ **do**
  - **if** $o(v) = \infty$ **then**
    - $\textsc{DFSExplore}(G, v)$

procedure $\textsc{DFSExplore}(G, v)$
- S.push(v)
- **while** S is not empty **do**
  - $v = S$.pop()
  - **if** $o(v) = \infty$ **then**
    - $o(v) \leftarrow i$;
    - $i \leftarrow i + 1$
    - **for** each $u \in vE$ **do**
      - S.push(u)

DFS Algorithm

Algorithm 1 Depth First Search

Input: Graph $G = (V, E)$

procedure $\mathrm{DFS}(G)$
- **for** each vertex $v \in V$ **do**
  - $o(v) \leftarrow \infty$
- $S \leftarrow$ EmptyStack()
- $i \leftarrow 1$
- **for** each vertex $v \in V$ **do**
  - **if** $o(v) = \infty$ **then**
    - $\textsc{DFSExplore}(G, v)$

procedure $\textsc{DFSExplore}(G, v)$
- S.push(v)
- **while** S is not empty **do**
  - $v = S$.pop()
  - **if** $o(v) = \infty$ **then**
    - $o(v) \leftarrow i$;
    - $i \leftarrow i + 1$
    - **for** each $u \in vE$ **do**
      - S.push(u)

Running time: $\mathcal{O}(|V| + |E|)$

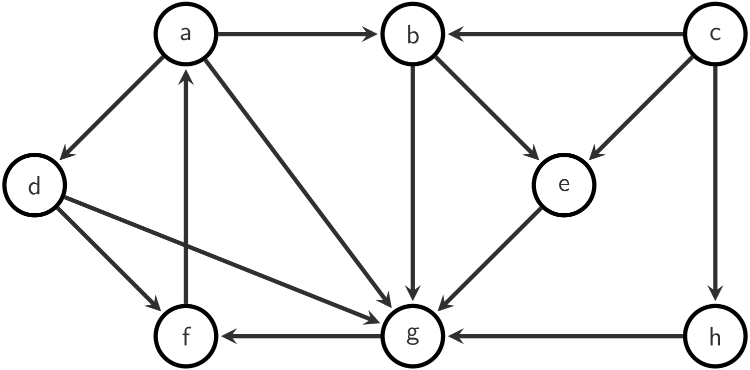
Cycle Detection using DFS

DFS can be modified to detect cycles: return `true` iff a "back-edge" is found (note that for undirected graphs, this back-edge must not be the edge from the predecessor).

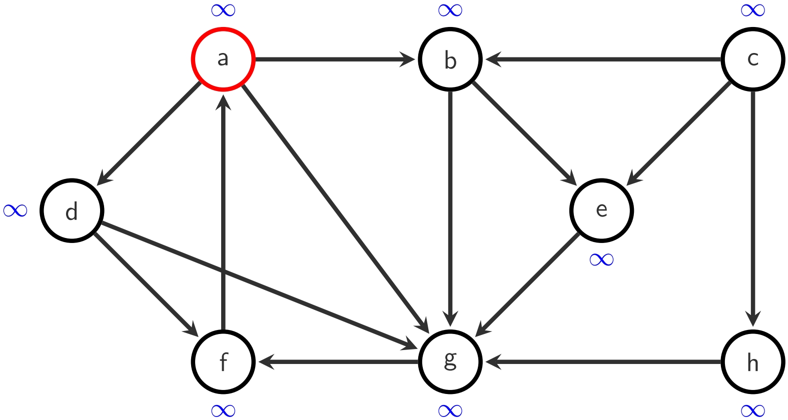
Breadth First Search (BFS)

Replacing worklist stack by a queue results in *Breadth First Search* (BFS).

Breadth First Search (BFS)

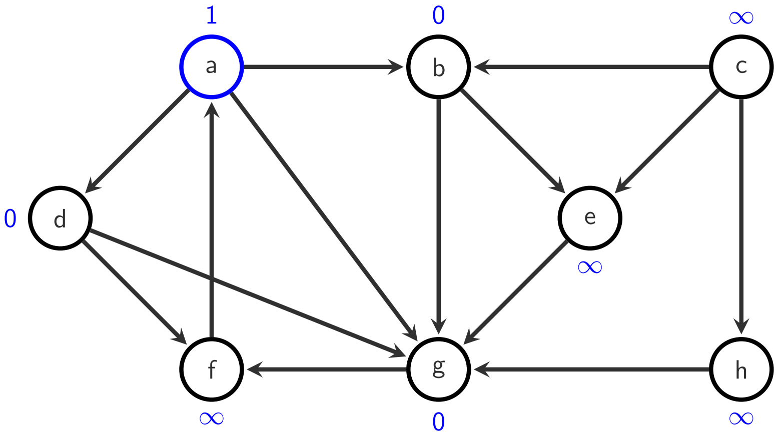


Breadth First Search (BFS)



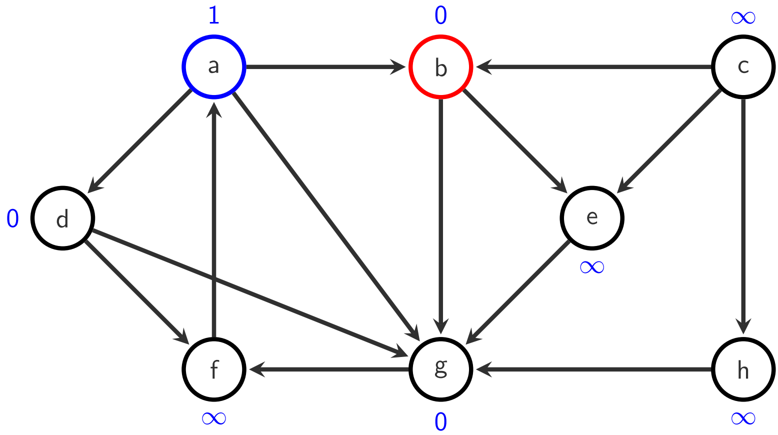
$S = [a]$

Breadth First Search (BFS)



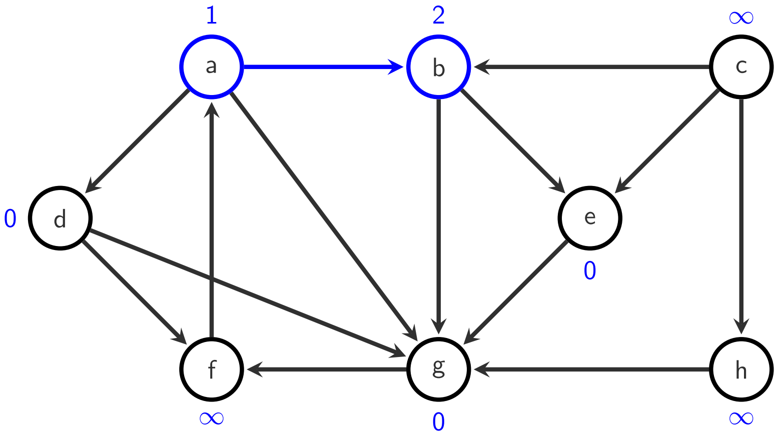
$S = [\mathrm{b}, \mathrm{d}, \mathrm{g}]$

Breadth First Search (BFS)



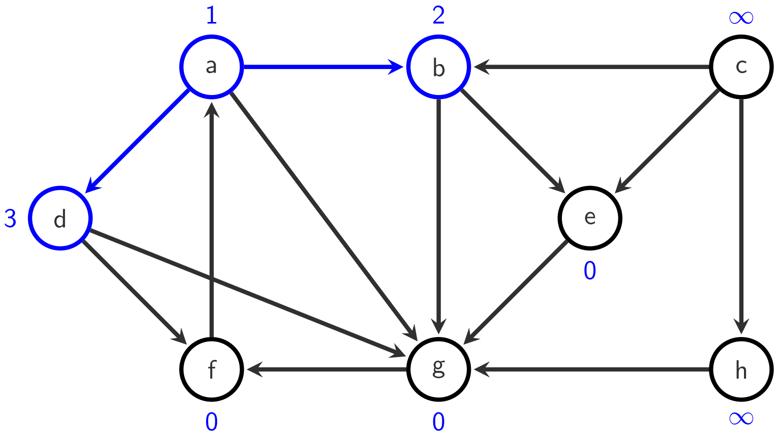
$S = [\text{b}, \text{d}, \text{g}]$

Breadth First Search (BFS)



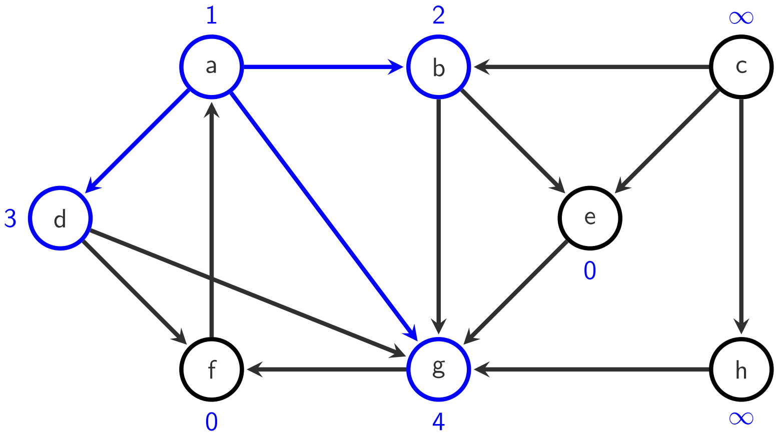
$S = [\mathrm{d}, \mathrm{g}, \mathrm{e}]$

Breadth First Search (BFS)



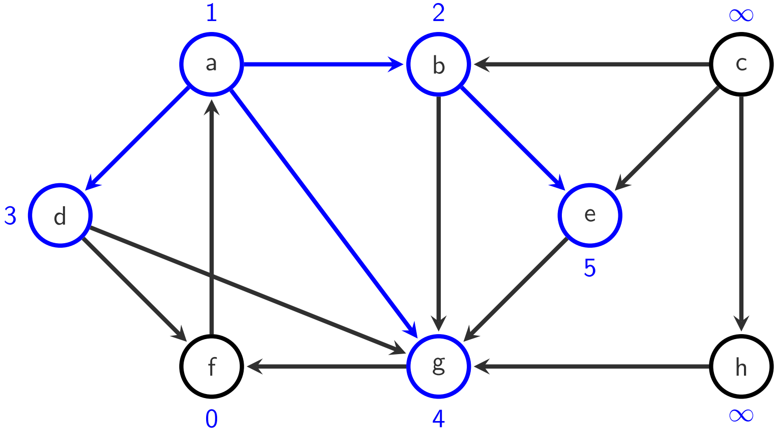
$S = [\mathrm{g}, \mathrm{e}, \mathrm{f}]$

Breadth First Search (BFS)



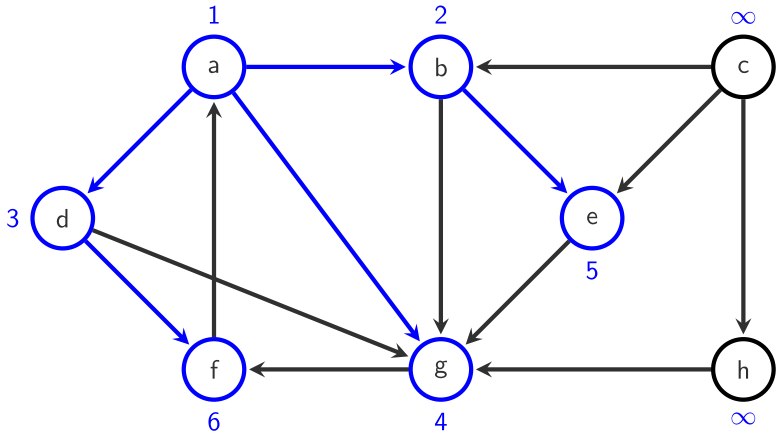
$S = [\mathrm{e}, \mathrm{f}]$

Breadth First Search (BFS)



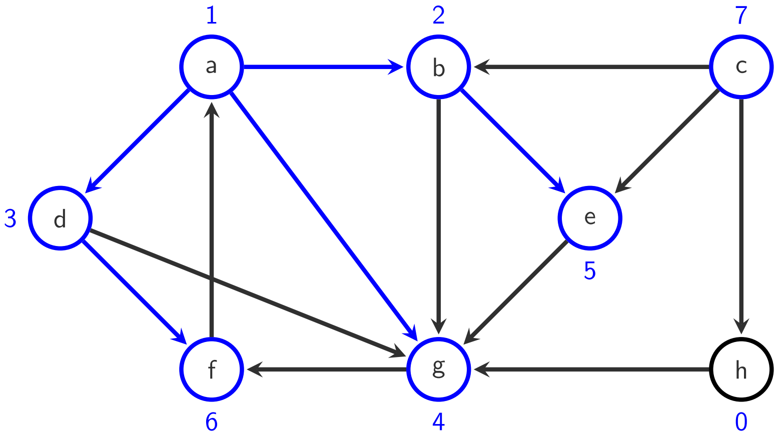
$S = [\mathrm{f}]$

Breadth First Search (BFS)



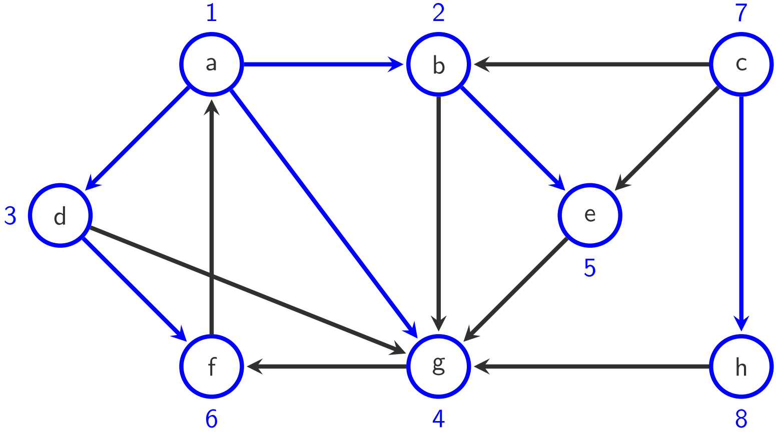
$S = []$

Breadth First Search (BFS)



$S = [\mathrm{h}]$

Breadth First Search (BFS)



$S = []$

BFS Algorithm

Algorithm 2 Breadth First Search

Input: Graph $G = (V, E)$

```
procedure BFS(G)
    for each vertex v ∈ V do
        o(v) ← ∞
    S ← EmptyQueue()
    i ← 1
    for each vertex v ∈ V do
        if o(v) = ∞ then
            BFSExplore(G, v)

procedure BFSExplore(G, v)
    S.enqueue(v)
    while S is not empty do
        v = S.dequeue()
        o(v) ← i;
        i ← i + 1
        for each u ∈ vE do
            if o(u) = ∞ then
                o(u) ← 0;
                S.enqueue(u)
```

BFS Algorithm

Algorithm 2 Breadth First Search

```
Input: Graph G = (V, E)
  procedure BFS(G)
      for each vertex v ∈ V do
          o(v) ← ∞
      S ← EmptyQueue()
      i ← 1
      for each vertex v ∈ V do
          if o(v) = ∞ then
              BFSExplore(G, v)

procedure BFSExplore(G, v)
    S.enqueue(v)
    while S is not empty do
        v = S.dequeue()
        o(v) ← i;
        i ← i + 1
        for each u ∈ vE do
            if o(u) = ∞ then
                o(u) ← 0;
                S.enqueue(u)
```

Running time: $\mathcal{O}(|V| + |E|)$

BFS: Remarks

- Note that vertices in worklist are automatically ordered by distance to source state.
- One can also keep track of the distances to the source state explicitly.
- The shortest paths can be obtained by keeping track of predecessors.
- Well-known generalization for directed graphs with non-negative weights: Dijkstra (essentially priority queue instead of FIFO queue for worklist)

Topological sort (TS)

Topological order

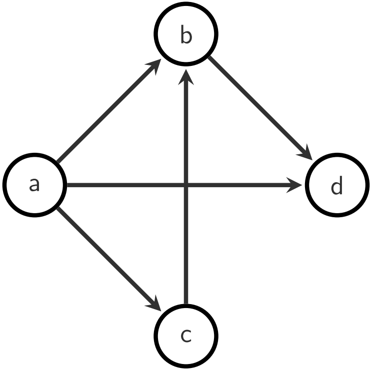
For a directed graph $G = (V, E)$, a *topological order* is an assignment $o : V \to \mathbb{N}$ such that for all $(u, v) \in E$, we have $o(u) < o(v)$.

- Topological order exists *if and only if* graph is acyclic (i.e. a DAG).
- Topological order may not be unique.
- Topological sort: Problem of finding a topological order.

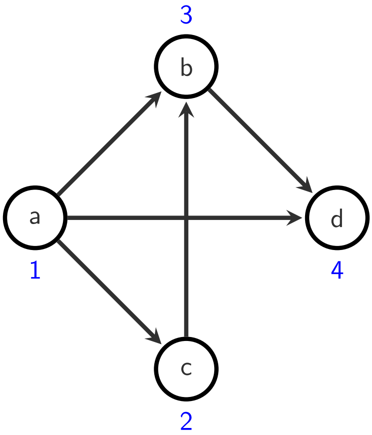
Usages

- Resolving dependencies.
- Instruction/task scheduling.
- Detecting cycles.
- Find shortest paths from a source in some weighted DAG in linear time.

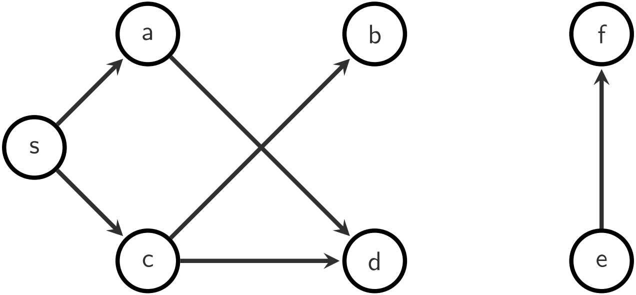
Topological order: Examples



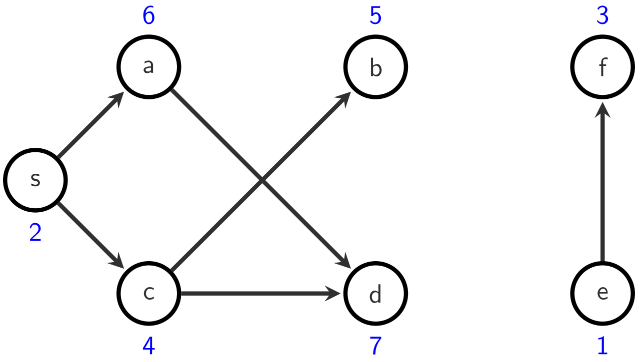
Topological order: Examples



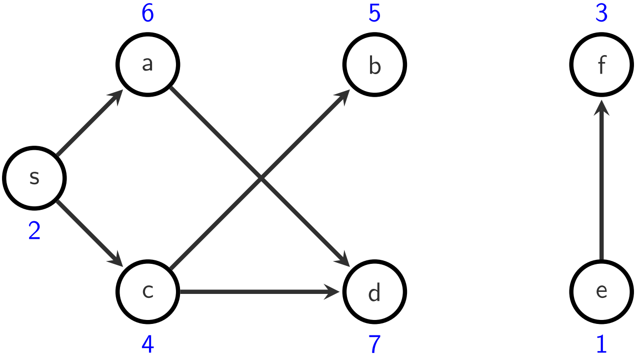
Topological order: Examples



Topological order: Examples



Topological order: Examples



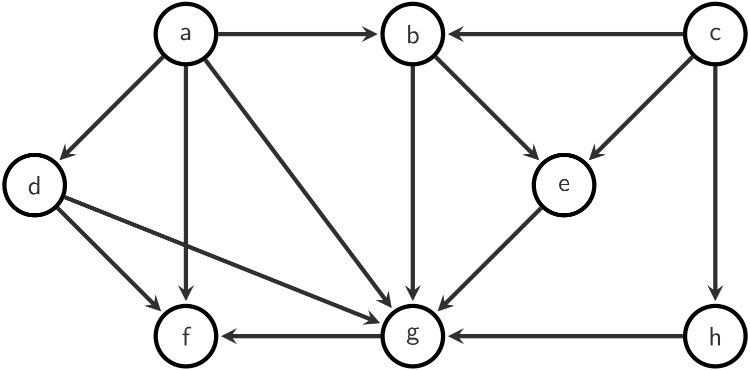
→ not unique!

TS

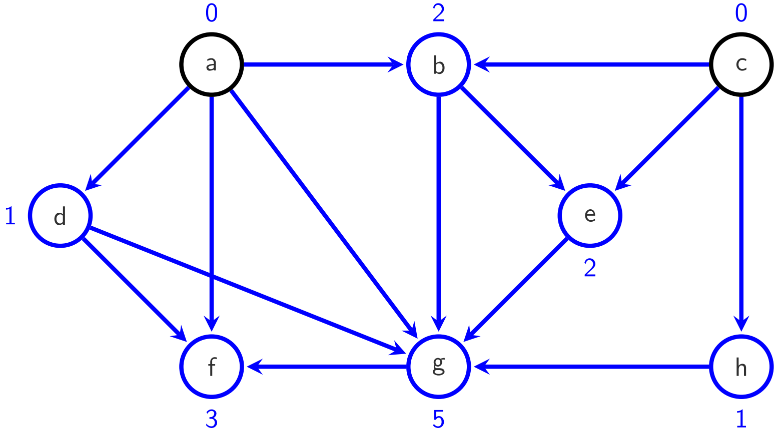
Idea:

1. For every node, store the number of predecessors.
2. Choose a node with 0 predecessors and remove it from the graph.
3. Repeat until no nodes with 0 predecessors left.
4. $\Rightarrow$ The order in which the nodes are removed is *topological*

TS (example)

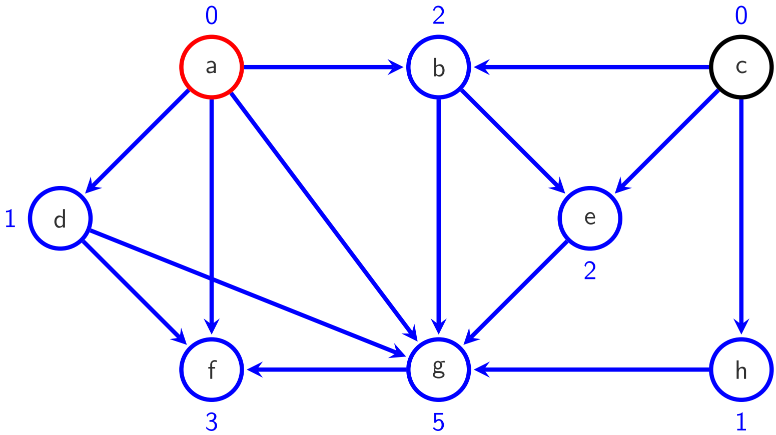


TS (example)



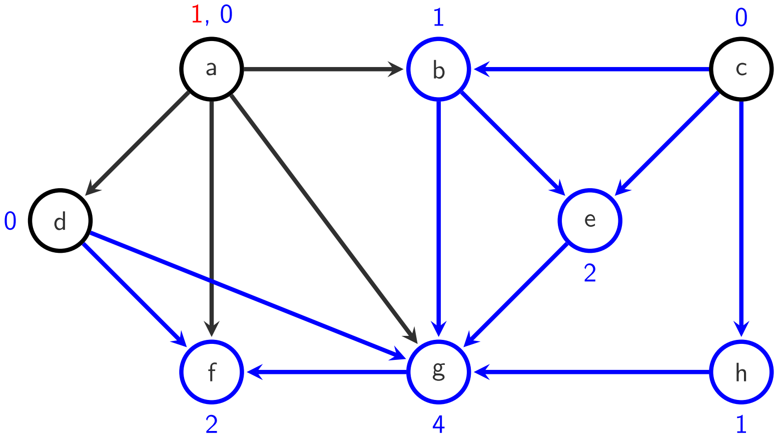
$S = [\mathrm{a}, \mathrm{c}]$

TS (example)



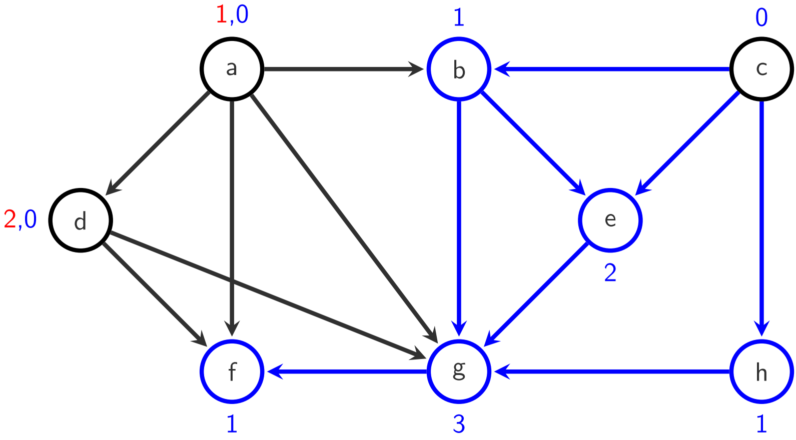
$S = [a, c]$

TS (example)



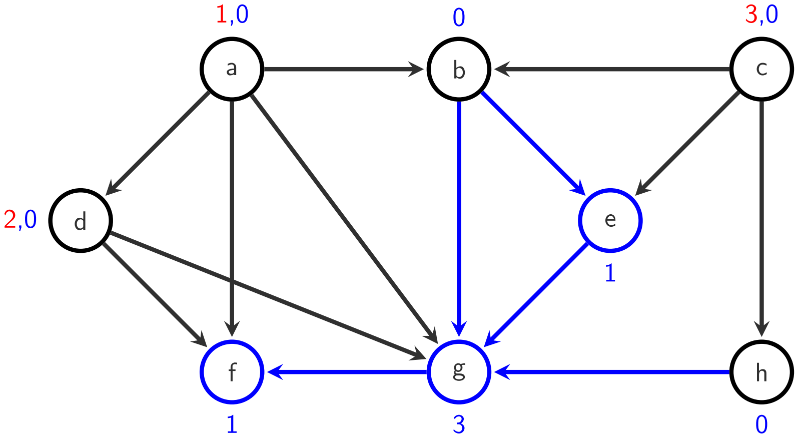
$S = [\mathrm{c}, \mathrm{d}]$

TS (example)



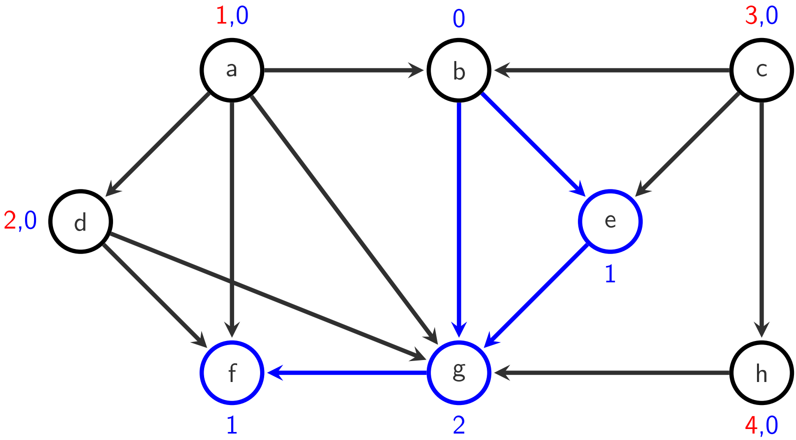
$S = [\mathrm{c}]$

TS (example)



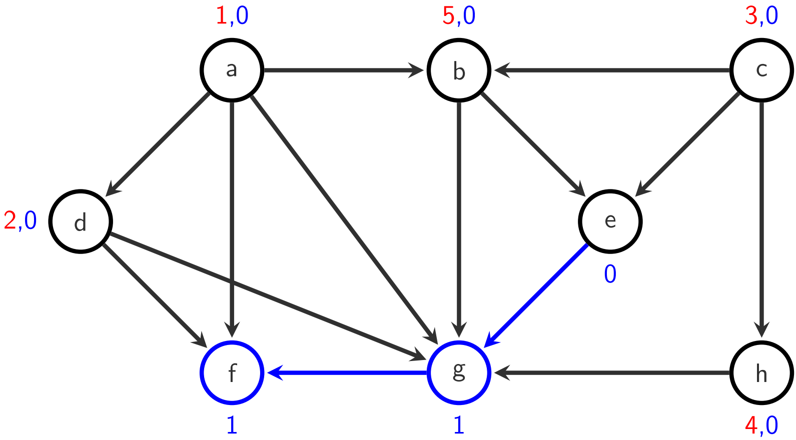
$S = [\mathrm{b}, \mathrm{h}]$

TS (example)



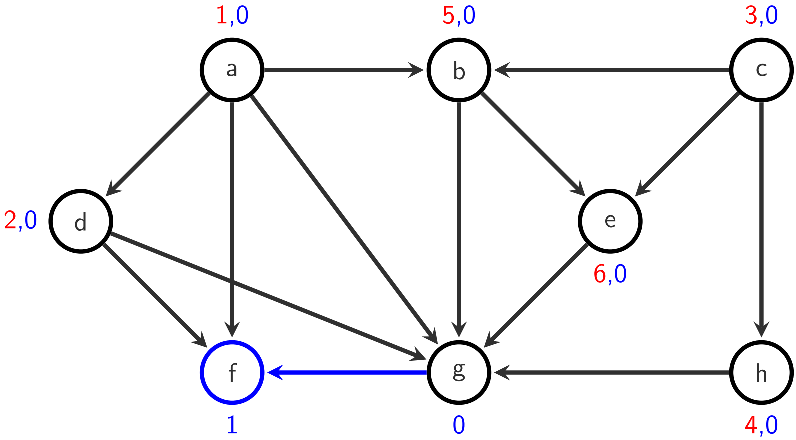
$S = [\text{b}]$

TS (example)



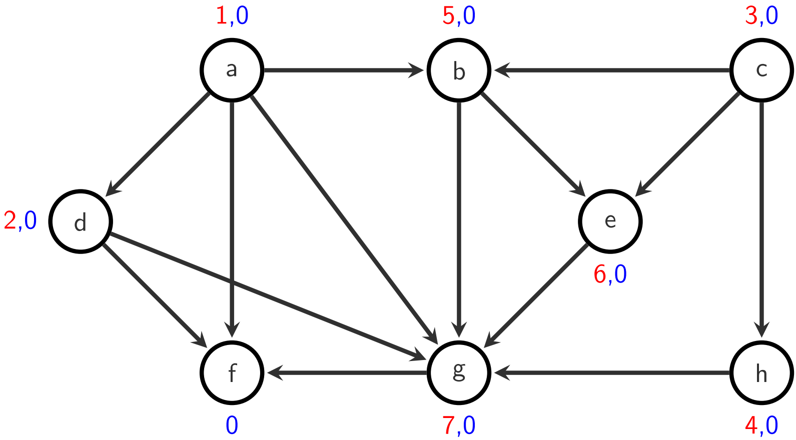
$S = [\text{e}]$

TS (example)



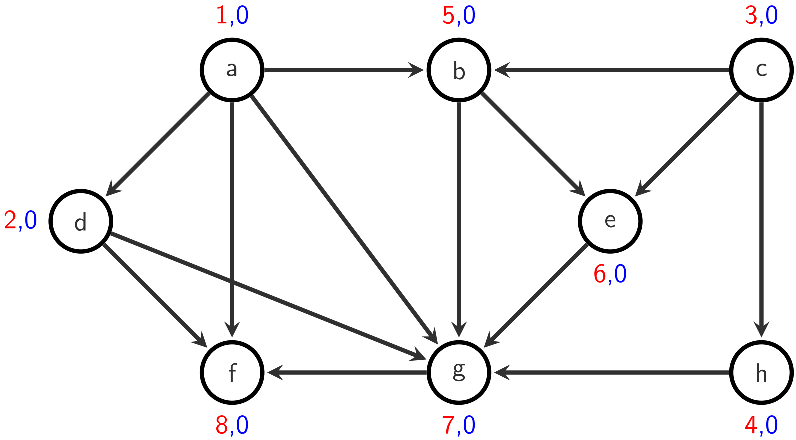
$S = [\mathrm{g}]$

TS (example)



$S = [\mathrm{f}]$

TS (example)



$S = []$

Algorithm 3 Topological sort

Input: Directed graph $G = (V, E)$

procedure $\textsc{TS}(G)$
- **for** each vertex $v \in V$ **do**
  - $o(v) \leftarrow \infty$
  - ▷ count predecessors
  - $pre(v) \leftarrow |\{u \mid v \in uE\}|$
- $S \leftarrow$ EmptyQueue()
- $i \leftarrow 1$
- **for** each vertex $v \in V$ **do**
  - **if** $pre(v) = 0$ **then**
    - $\textsc{TSExplore}(G, v)$

procedure $\textsc{TSExplore}(G, v)$
- **if** $o(v) = \infty$ **then**
  - S.push(v)
- **while** S is not empty **do**
  - $v = S$.pop()
  - $o(v) \leftarrow i$; $i \leftarrow i + 1$
  - **for** each $u \in vE$ **do**
    - $pre(u) \leftarrow pre(u) - 1$
    - **if** $pre(u) = 0$ **then**
      - S.push(u)

Algorithm 3 Topological sort

Input: Directed graph $G = (V, E)$

procedure $\text{TS}(G)$
- **for** each vertex $v \in V$ **do**
  - $o(v) \leftarrow \infty$
  - ▷ count predecessors
  - $pre(v) \leftarrow |\{u \mid v \in uE\}|$
- $S \leftarrow$ EmptyQueue()
- $i \leftarrow 1$
- **for** each vertex $v \in V$ **do**
  - **if** $pre(v) = 0$ **then**
    - $\text{TSExplore}(G, v)$

procedure $\text{TSExplore}(G, v)$
- **if** $o(v) = \infty$ **then**
  - S.push(v)
- **while** S is not empty **do**
  - $v = S$.pop()
  - $o(v) \leftarrow i$; $i \leftarrow i + 1$
  - **for** each $u \in vE$ **do**
    - $pre(u) \leftarrow pre(u) - 1$
    - **if** $pre(u) = 0$ **then**
      - S.push(u)

The graph is cyclic iff unvisited vertices ($v \in V$ with $o(v) = \infty$) remain.

Algorithm 3 Topological sort

Input: Directed graph $G = (V, E)$

procedure $\mathrm{TS}(G)$
  for each vertex $v \in V$ **do**
    $o(v) \leftarrow \infty$
    ▷ count predecessors
    $pre(v) \leftarrow |\{u \mid v \in uE\}|$
  $S \leftarrow \text{EmptyQueue}()$
  $i \leftarrow 1$
  for each vertex $v \in V$ **do**
    if $pre(v) = 0$ **then**
      $\textsc{TSExplore}(G, v)$

procedure $\textsc{TSExplore}(G, v)$
  if $o(v) = \infty$ **then**
    $S.\text{push}(v)$
  while S is not empty **do**
    $v = S.\text{pop}()$
    $o(v) \leftarrow i$; $i \leftarrow i + 1$
    for each $u \in vE$ **do**
      $pre(u) \leftarrow pre(u) - 1$
      if $pre(u) = 0$ **then**
        $S.\text{push}(u)$

Running time: $\mathcal{O}(|V| + |E|)$
(For that, need to count predecessors in linear time...)

TS: Remarks

Another possible way of implementing TS (again in linear time) is using DFS:

- Run modified version, where instead of the discovery time, the *finish* time of each vertex is returned
- A node *finished* if all its successors have been removed from the FIFO queue

SCC Discovery

SCC Discovery

Not in the scope of our course, but still worth mentioning:

SCC Discovery

Given a graph $G = (V, E)$, return

1. all SCCs and
2. (optionally) a topological order on the SCCs

- This problem can still be solved in $\mathcal{O}(|V| + |E|)$
- One algorithm: Tarjan's SCC Discovery
- Another algorithm:
  1. Run DFS, recording finish times
  2. Run DFS in G^T, where G^T is G but with all edges reversed, and in the main loop (where `DFSExplore` is called) consider vertices in decreasing finish time order of first DFS
  3. The SCCs are the vertices of each tree explored in the second DFS search

Minimum spanning trees

Minimum spanning trees (MST)

Let $G = (V, E)$ be a graph. A graph (V', E') with $V' \subseteq V$, $E' \subseteq E$ is called *subgraph* of G. It is *spanning* if $V' = V$. Sometimes, we identify a spanning subgraph (V, E') with its edge set E'.

Spanning tree

For an undirected graph $G = (V, E)$, a *spanning tree* of G is a spanning subgraph which is a tree.

Weighted graphs

We now consider graphs with a *weight function* $w : E \to \mathbb{R}$ on the edges. For a subset of edges $E' \subseteq E$, we define $w(E') := \sum_{e \in E'} w(e)$.

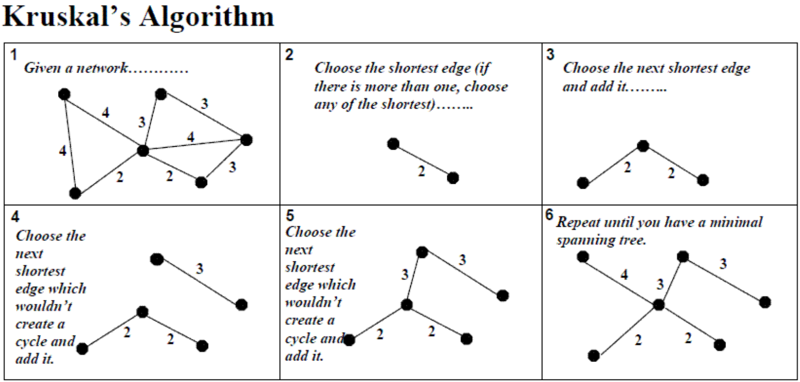
Minimum (weight) spanning tree

For an undirected graph $G = (V, E)$ with a weight function $w : E \to R$, a *minimum spanning tree (MST)* is a spanning tree S of G such that for all spanning trees T of G, we have $w(S) \leq w(T)$.

Minimum spanning trees (MST)

- Spanning trees only exist for connected graphs.
- Otherwise, a spanning tree exists for each connected component.
- All spanning trees of a graph have the same number of edges.
- Negative weights can be avoided by adding a constant to all weights.
- Maximum spanning tree can be obtained with $w'(e) = -w(e)$.

Kruskal and Prim



Algorithm 4 Kruskal's algorithm

Input: Undirected graph $G = (V, E)$

procedure $\textsc{Kruskal}(G)$
- $S \leftarrow \emptyset$ ▷ Current set of edges
- $L \leftarrow$ List of edges $e \in E$ sorted in increasing order by $w(e)$
- $U \leftarrow$ Union-Find structure initialized over set V
- **for** each edge (u, v) in L in order **do**
  - ▷ Test if vertices are in different components
  - **if** $U.\text{find}(u) \neq U.\text{find}(v)$ **then**
    - ▷ If yes, add edge to MST and merge components
    - $U.\text{union}(u, v)$
    - $S \leftarrow S \cup \{e\}$

Iff vertices in different components remain, the graph is not connected. (Note that in that case, a minimum spanning *forest* is constructed.)

Remark: Kruskal's algorithm is an instance of a *greedy algorithm*. Greedy algorithms can be shown to be correct for (and only for) *matroids*.

Analysis of Kruskal's algorithm

Running time

- Sorting of edges: $\mathcal{O}(|E| \log |E|)$
- With α as the inverse Ackermann function, i.e. $\alpha = f^{-1}$ with $f(n) = A(n, n)$:
- $2|E|$ find operations: $\mathcal{O}(|E| \alpha(|V|))$
- $|V|$ union operations: $\mathcal{O}(|V| \alpha(|V|))$
- In total: $\mathcal{O}(|E| \log |E|)$

Proof of correctness for Kruskal's algorithm

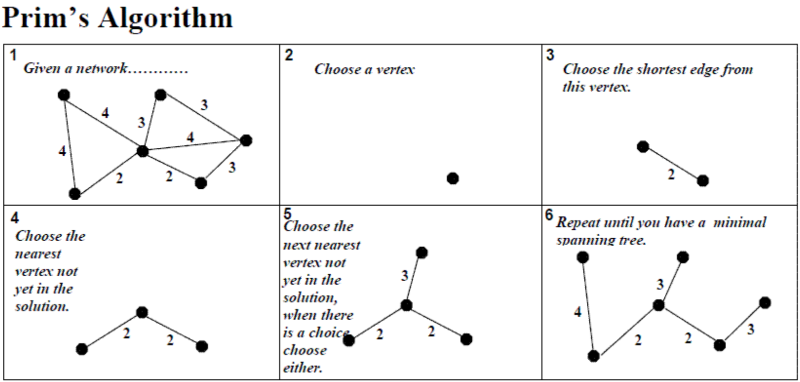
Lemma

Let $T \subseteq E$ be a set of edges such that there is a minimum spanning tree S of G with $T \subseteq S$.
Let $e \in E \setminus T$ be an edge such that $T \cup \{e\}$ does not create a cycle, with e having minimal weight among all of these edges.
Then, there is a minimum spanning tree S' of G such that $T \cup \{e\} \subseteq S'$.

Proof.

When $e \in S$, then $S' := S$ fulfills the requirement.
When $e \notin S$, then $S \cup \{e\}$ has a cycle c, and there is an edge $f \neq e$ in c that is not in T (otherwise adding e to T would create a cycle). Then $S' := S \setminus \{f\} \cup \{e\}$ is a also a spanning tree, and $w(S') \leq w(S)$, as $w(e) \leq w(f)$. Hence, as S is a minimum spanning tree, S' is also a minimium spanning tree.

Kruskal and Prim



Prim: Implementation

- Use 3 colors for the vertices:
    - *black* (finished node — already part of the MST)
    - *grey* (discovered node – at least one connection to a black node)
    - *white* (unknown node — no connection to initial node found yet)

1. Start at a single node, keep track of encountered nodes.
2. Choose a grey node that is closest to any black node, color it black and all its white neighbors grey.
3. Repeat until no grey nodes are left.

Algorithm 5 Prim's algorithm

```
Input: Graph G = (V, E)
  procedure Prim(G)
      S ← ∅
      for each vertex v ∈ V do
          visited(v) ← false
          c(v) ← ∞
      PQ ← PriorityQueue over V
      s ← any v ∈ V
      PrimVisit(s)
      while PQ is not empty do
          v ← PQ.deleteMin()
          S ← S ∪ {{pre(v), v}}
          PrimVisit(v)

procedure PrimVisit(v)
    visited(v) ← true
    for each u ∈ vE do
        if not visited(u) then
            if w(v, u) < c(u) then
                pre(u) ← v
                c(u) ← w(v, u)
                if u in PQ then
                    PQ.decreaseKey(u, c(u))
                else
                    PQ.insert(u, c(u))
```

If not all vertices were visited, the graph is not connected.

Analysis of Prim's algorithm

Running time

- Graph exploration without priority queue: $\mathcal{O}(|V| + |E|)$
- With Fibonacci heap as priority queue:
- $|V|$ insert operations: $\mathcal{O}(|V|)$
- $|E|$ decreaseKey operations: $\mathcal{O}(|E|)$
- $|V|$ deleteMin operations: $\mathcal{O}(|V| \log |V|)$
- In total: $\mathcal{O}(|E| + |V| \log |V|)$

Note: Fibonacci Heaps not part of the standard library of C++/Rust. However, libraries exist, e.g. the `fibonacci_heap` crate (remember to cite the correct source when using it!).

Prim vs. Kruskal

Which one to choose?

- Prim: $\mathcal{O}(|E| + |V| \log |V|)$
- Kruskal: $\mathcal{O}(|E| \log |E|)$
- Prim looks asymptotically better
- However, in practice, usually Kruskal is faster, unless the graph is large and dense

Remark: Sometimes, people write Kruskal's running time as $\mathcal{O}(|E| \log |V|)$. Why?