

Algorithms for Programming Contests - Week 3

Tobias Meggendorfer, Philipp Meyer,
Christian Müller, Gregor Schwarz
`conpra@in.tum.de`

October 31, 2018

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 $E \subseteq \{\{u, v\} \mid u, v \in V\}$.

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 $i \in [1, n-1]$, and $v_i \neq v_j$ for all $i \neq j$.

- A graph is *cyclic* if there is a path $p = v_1 \dots v_n$ with $v_1 \in v_n E$, otherwise it is *acyclic*.
- An undirected graph is *connected* if for every pair of vertices $u, v \in V$, there is path from u to v .
- For an undirected graph, a *connected component* is a maximal set $V' \subseteq V$ where for all $u, v \in V'$, there is a path from u to v .
- 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 a *forest*, and each connected component is a tree.
- A directed acyclic graph is also called a *DAG*.

Graphs as an abstract data type

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 $(u, v) \in E$.

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 $u \in vE$.

Graph traversal

Graph traversal

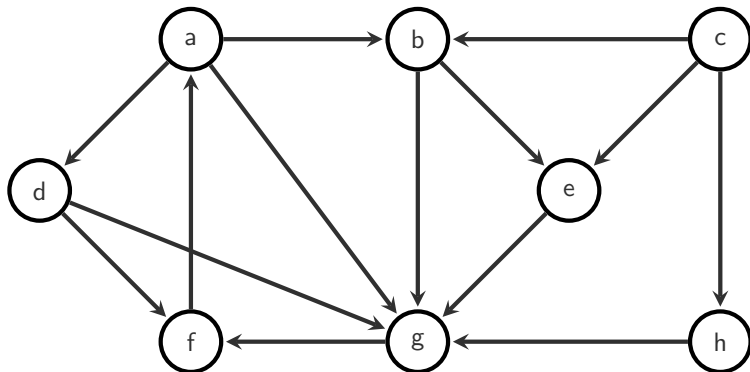
Graph traversal

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

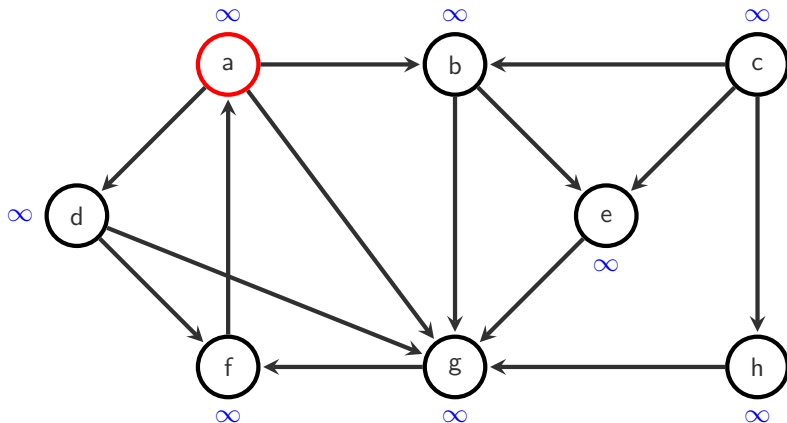
Usages

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

Depth First Search (DFS)

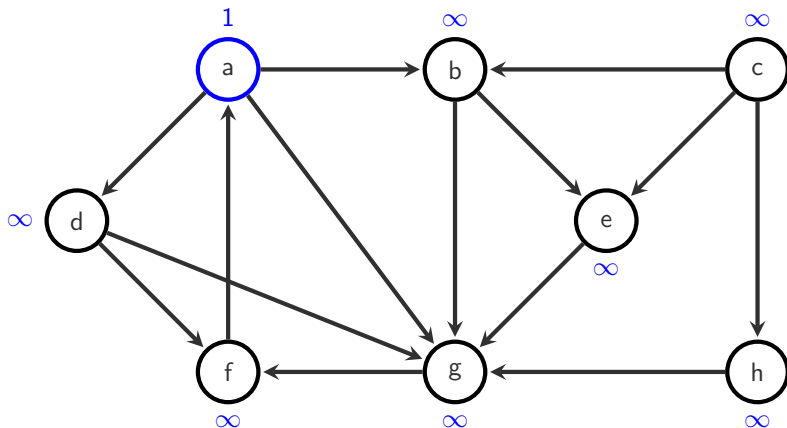


Depth First Search (DFS)



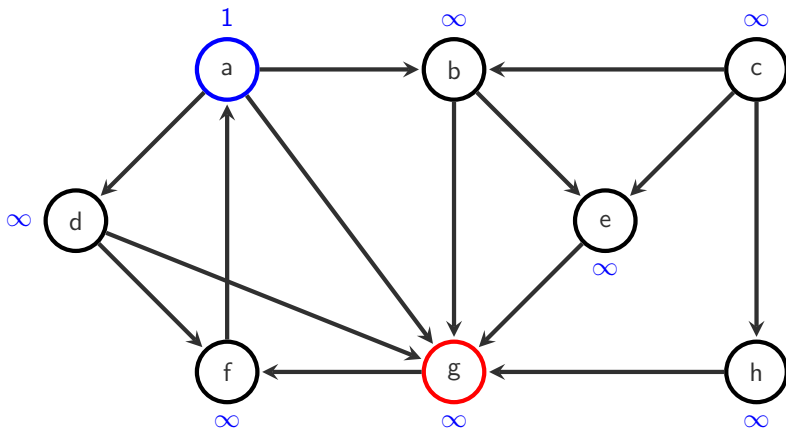
$$S = [a]$$

Depth First Search (DFS)



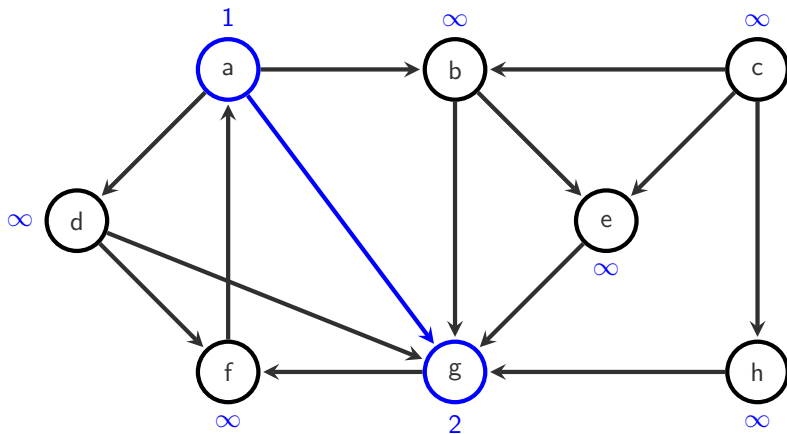
$$S = [b, d, g]$$

Depth First Search (DFS)



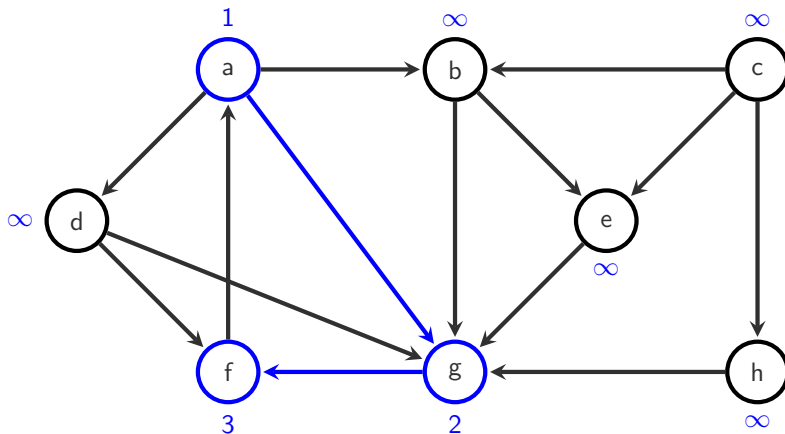
$$S = [b, d, g]$$

Depth First Search (DFS)



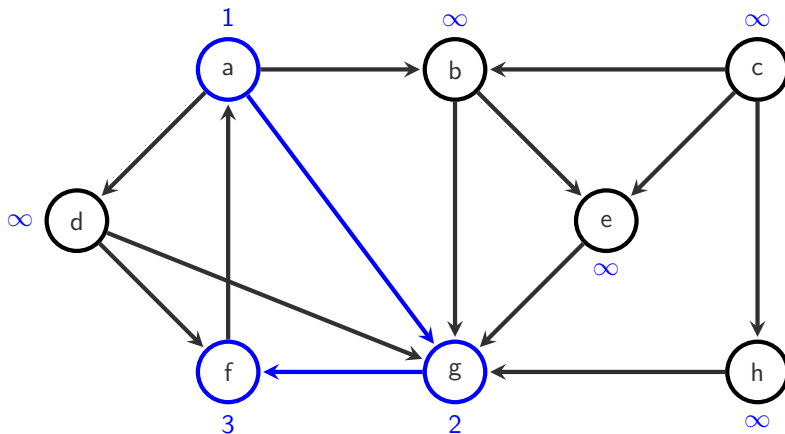
$$S = [b, d, f]$$

Depth First Search (DFS)



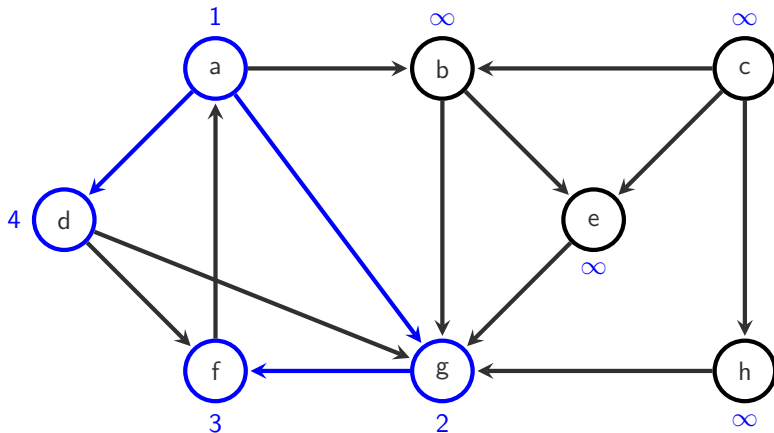
$S = [b, d, a]$

Depth First Search (DFS)



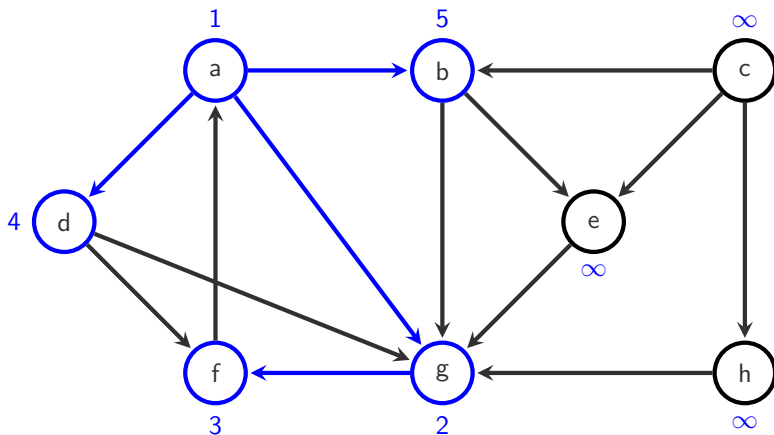
$$S = [b, d]$$

Depth First Search (DFS)



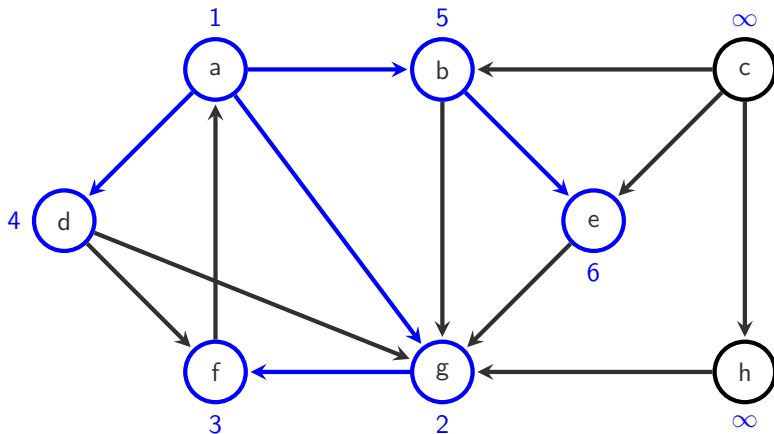
$$S = [b, f, g]$$

Depth First Search (DFS)



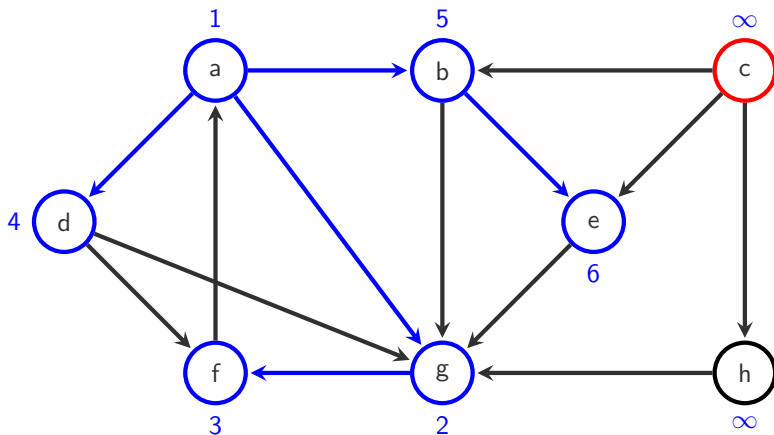
$$S = [e, g]$$

Depth First Search (DFS)

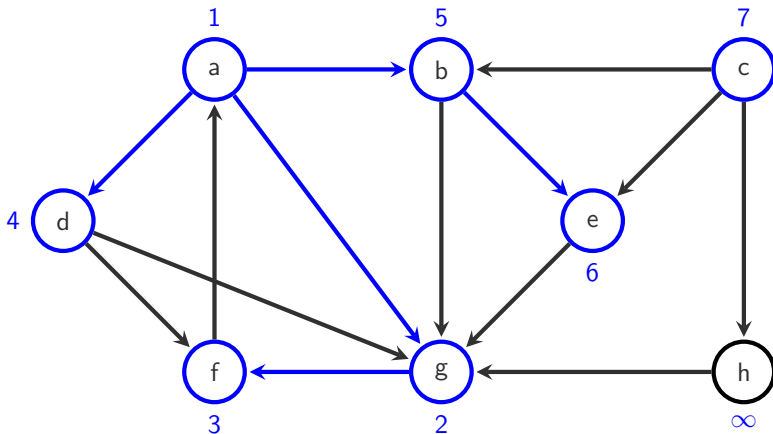


$$S = [g]$$

Depth First Search (DFS)

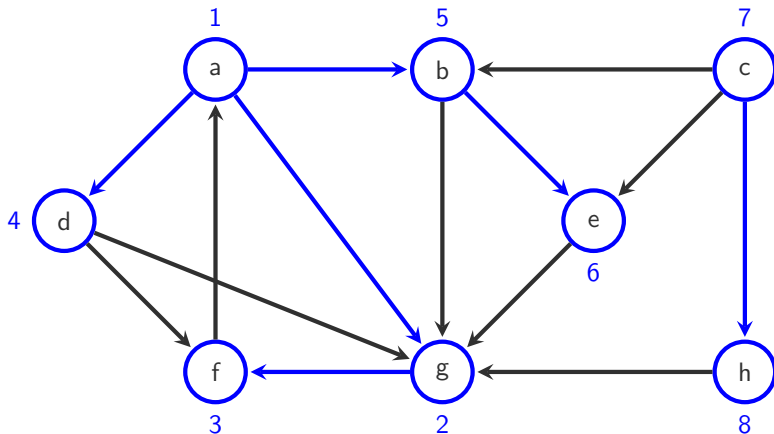
 $S = [c]$

Depth First Search (DFS)



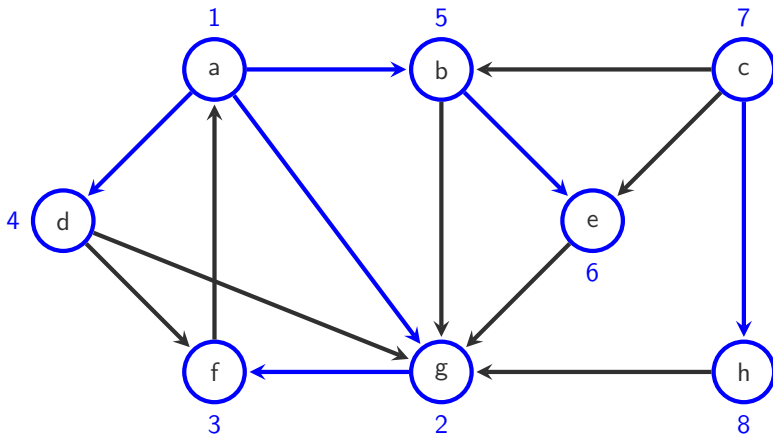
$$S = [e, h]$$

Depth First Search (DFS)



$$S = [e, g]$$

Depth First Search (DFS)



$S = []$

DFS Algorithm

Algorithm 1 Depth-first search

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

procedure DFS(G)

for each vertex $v \in V$ **do**

$o(v) \leftarrow \infty$

$S \leftarrow \text{EmptyStack}()$

$i \leftarrow 1$

for each vertex $v \in V$ **do**

if $o(v) = \infty$ **then**

 DFSEXPLORE(G, v)

procedure DFSEXPLORE(G, v)

$S.\text{push}(v)$

while S is not empty **do**

$v = S.\text{pop}()$

if $o(v) = \infty$ **then**

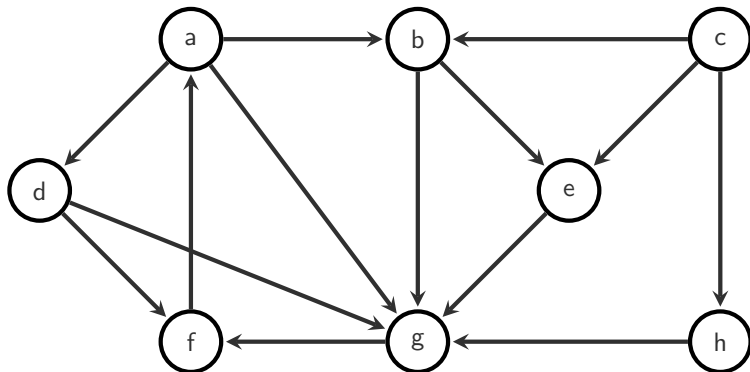
$o(v) \leftarrow i;$

$i \leftarrow i + 1$

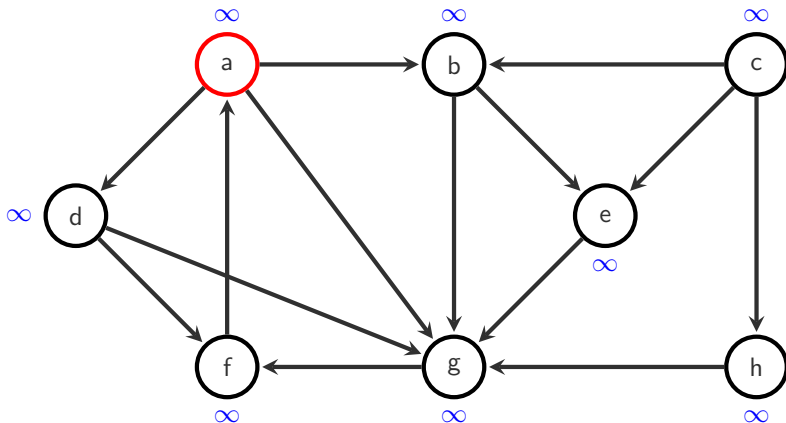
for each $u \in vE$ **do**

$S.\text{push}(u)$

Breadth First Search (BFS)

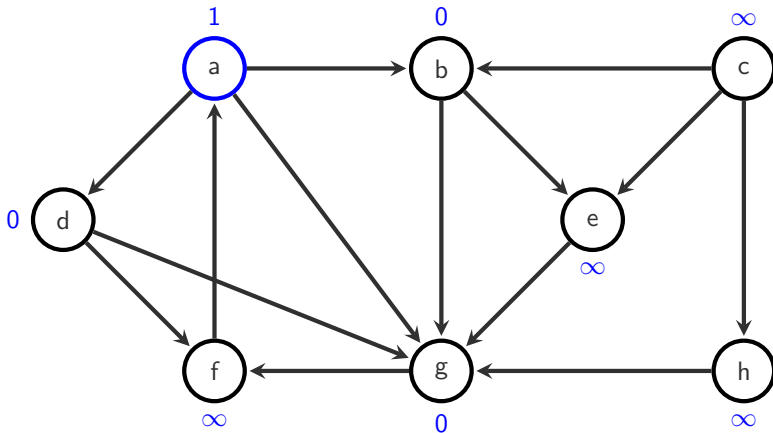


Breadth First Search (BFS)



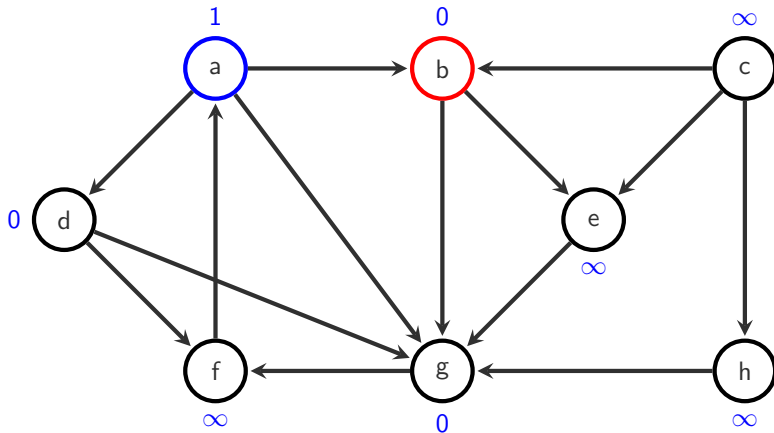
$$S = [a]$$

Breadth First Search (BFS)



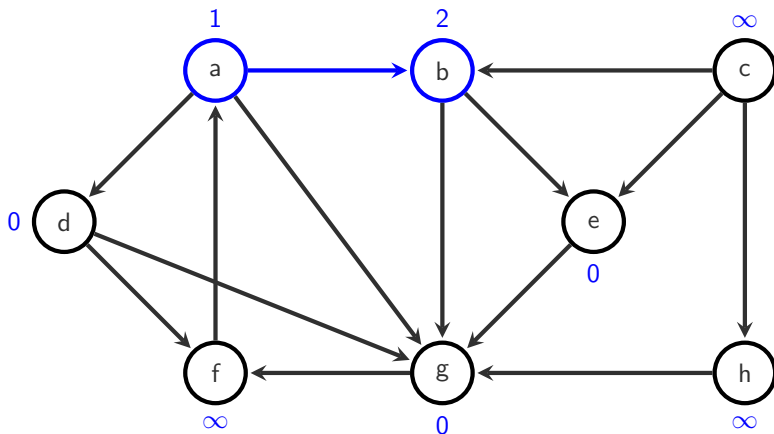
$$S = [b, d, g]$$

Breadth First Search (BFS)



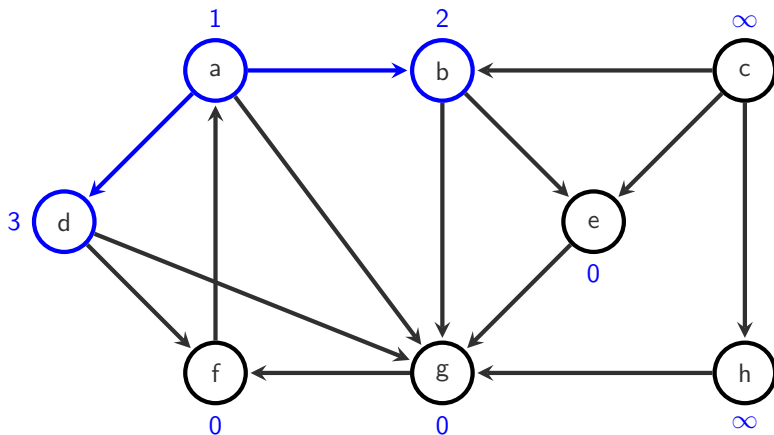
$$S = [\textcolor{red}{b}, d, g]$$

Breadth First Search (BFS)



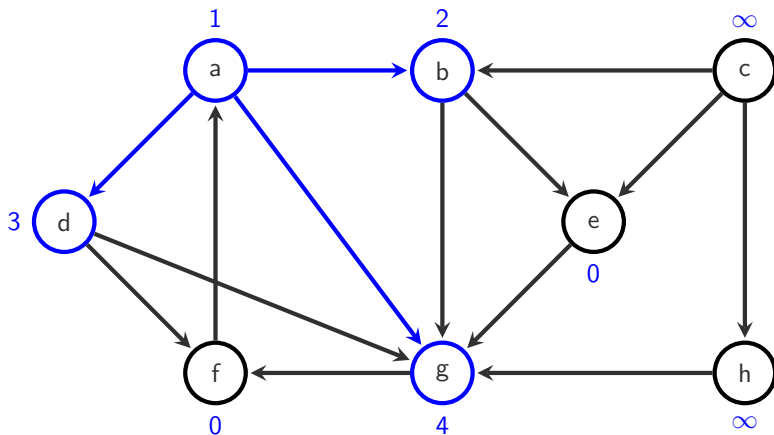
$S = [d, g, e]$

Breadth First Search (BFS)



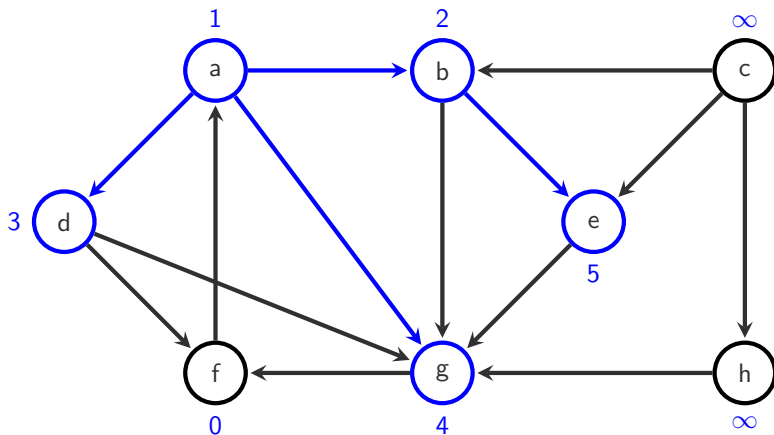
$$S = [g, e, f]$$

Breadth First Search (BFS)



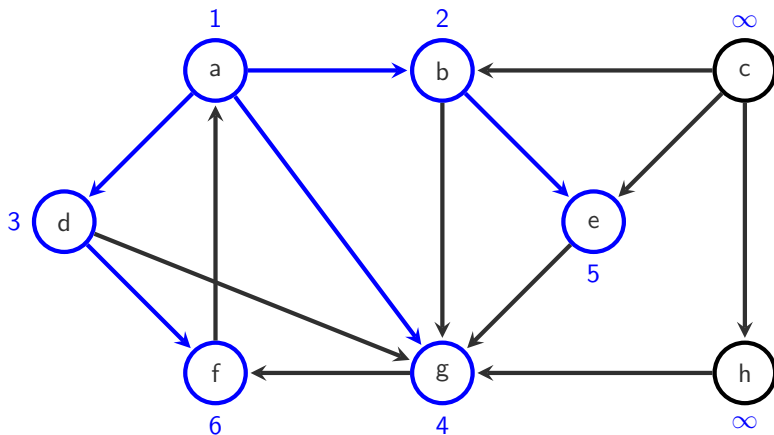
$$S = [e, f]$$

Breadth First Search (BFS)



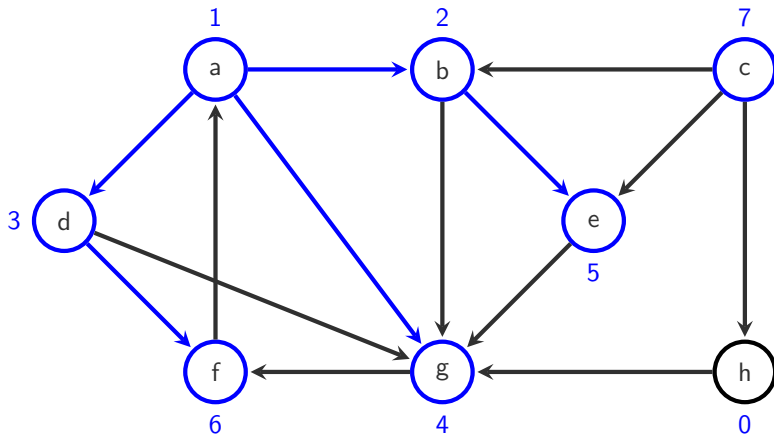
$$S = [f]$$

Breadth First Search (BFS)



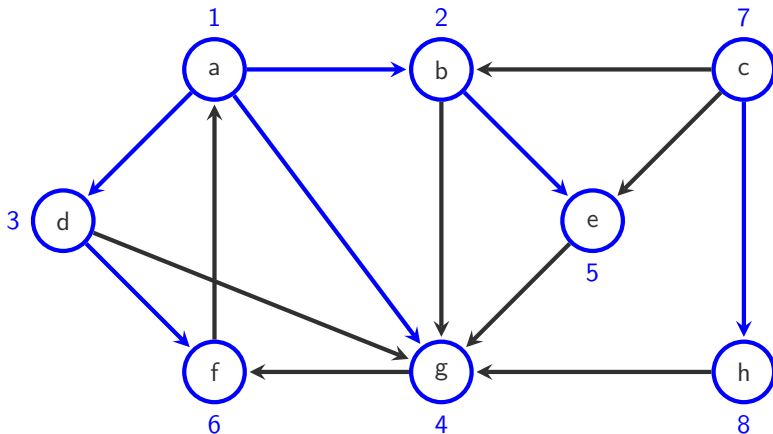
$S = []$

Breadth First Search (BFS)



$S = [h]$

Breadth First Search (BFS)



$S = []$

BFS Algorithm

Algorithm 2 Breadth-first search

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

procedure BFS(G)

for each vertex $v \in V$ **do**

$o(v) \leftarrow \infty$

$S \leftarrow \text{EmptyQueue}()$

$i \leftarrow 1$

for each vertex $v \in V$ **do**

if $o(v) = \infty$ **then**

 BFSEXPLOR(G, v)

procedure BFSEXPLOR(G, v)

$S.\text{enqueue}(v)$

while S is not empty **do**

$v = S.\text{dequeue}()$

$o(v) \leftarrow i$;

$i \leftarrow i + 1$

for each $u \in vE$ **do**

if $o(u) = \infty$ **then**

$o(u) \leftarrow 0$;

$S.\text{enqueue}(u)$

Topological sort (TS)

Topological order

For a directed graph $G = (V, E)$, a *topological order* is an assignment $o : V \rightarrow \mathbb{N}$ 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.

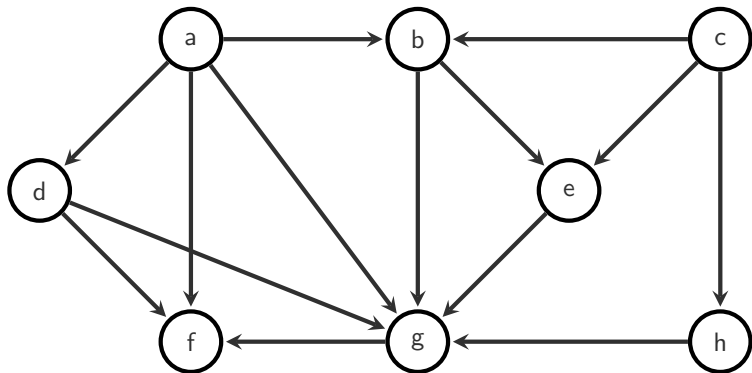
Usages

- Resolving dependencies.
- Instruction scheduling.
- Determine order for compilation multi-source programs.
- Detecting cycles.

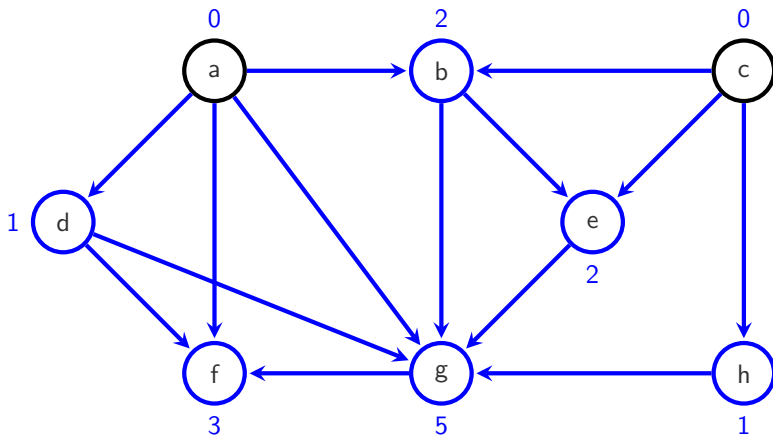
Topological sort (Implementation)

- 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)

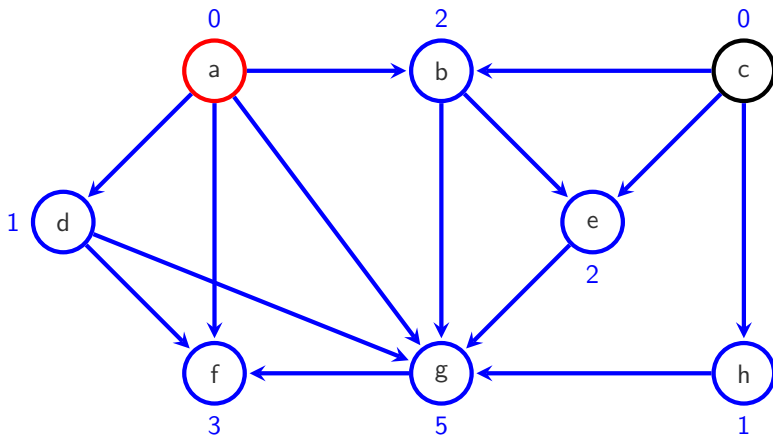


TS (example)



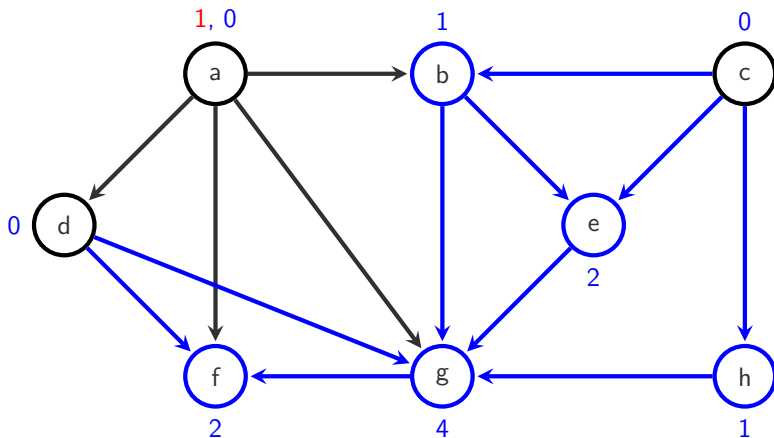
$$S = [a, c]$$

TS (example)



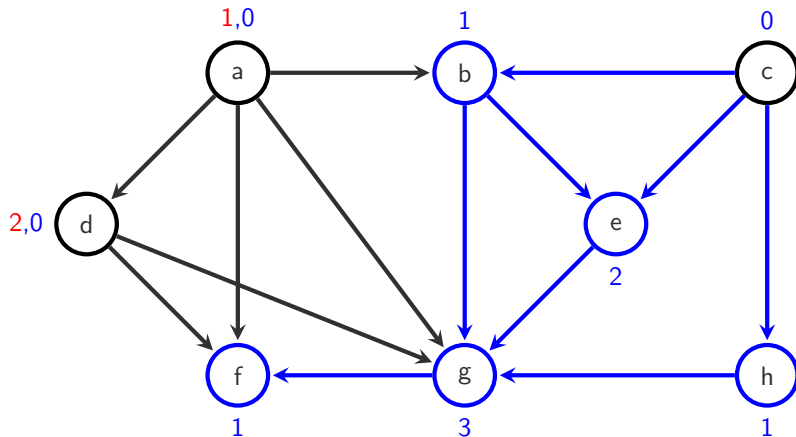
$$S = [a, c]$$

TS (example)



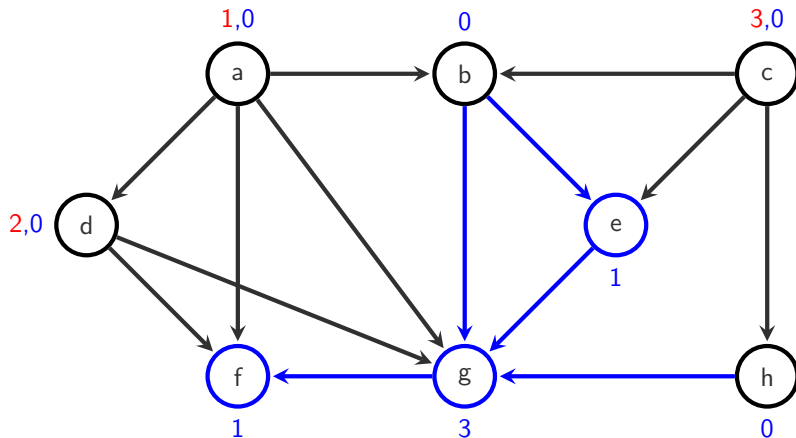
$$S = [c, d]$$

TS (example)



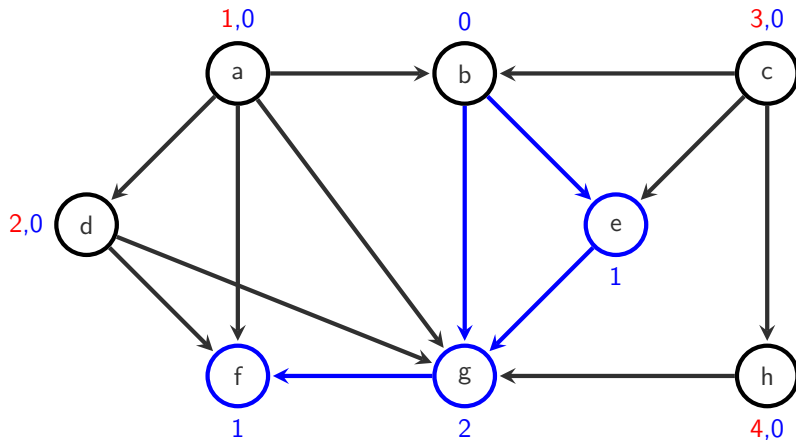
$S = [c]$

TS (example)



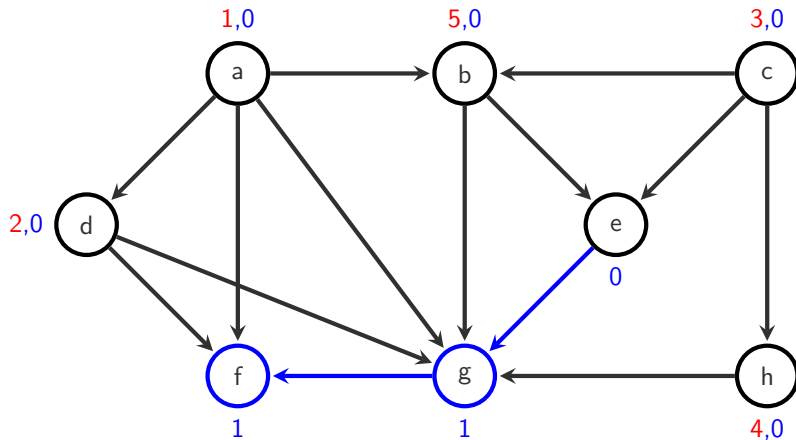
$$S = [b, h]$$

TS (example)



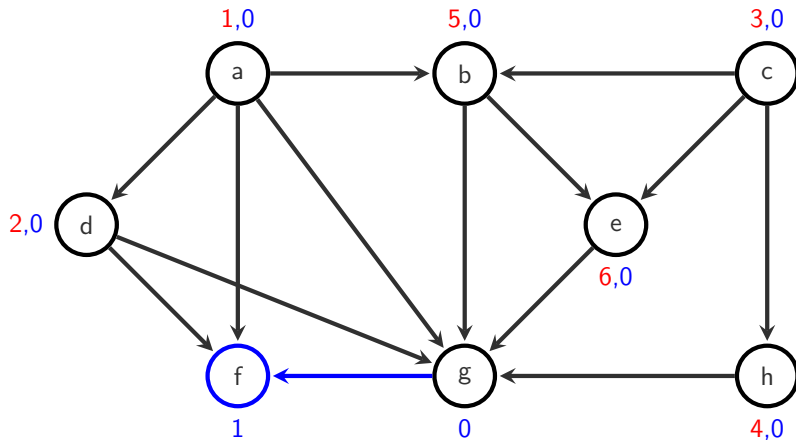
$$S = [b]$$

TS (example)



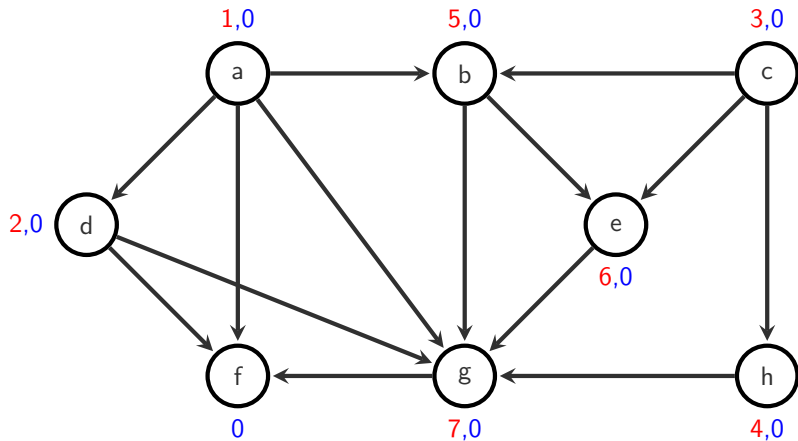
$$S = [e]$$

TS (example)



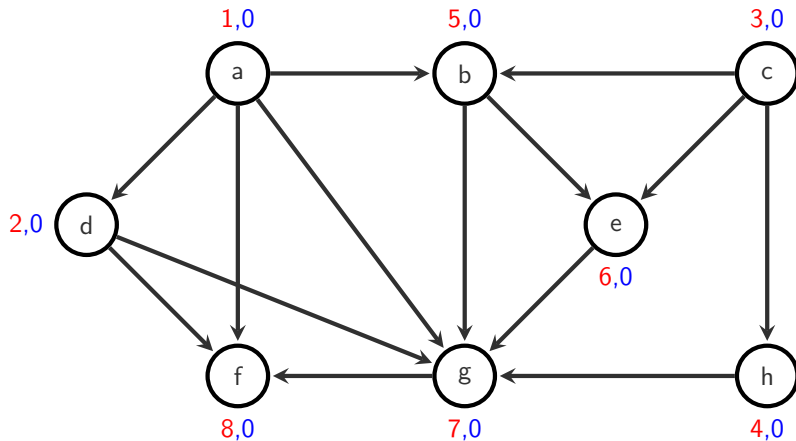
$$S = [g]$$

TS (example)



$$S = [f]$$

TS (example)



$S = []$

- └ Graph traversal
 - └ Topological sort

Algorithm 3 Topological sort

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

procedure 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**

 TSEXPLOR(G, v)

procedure TSEXPLOR(G, v)

$Q.\text{push}(v)$

while Q 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**

$S.\text{push}(u)$

If unvisited vertices with $o(v) = \infty$ remain, then the graph is cyclic.

Analysis of DFS, BFS and TS

Running time

- Each vertex is visited at most once: $\mathcal{O}(|V|)$
- For each vertex, each successor considered at most once:
 $\mathcal{O}(\sum_{v \in V} |vE|) = \mathcal{O}(|E|)$
- In total: $\mathcal{O}(|V| + |E|)$
- For topological sort, count number of predecessors in linear time.

Minimum spanning trees

Minimum spanning trees (MST)

Spanning tree

For an undirected graph $G = (V, E)$, a *spanning tree* of G is a subset of edges $T \subseteq E$ such that (V, T) forms a tree, i.e. is connected and acyclic.

Weighted graphs

We now consider graphs with a *weight function* $w : E \rightarrow \mathbb{R}$ 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

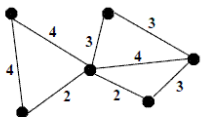

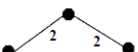
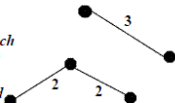
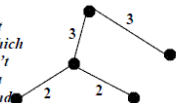
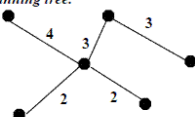
For an undirected graph $G = (V, E)$ with a weight function $w : E \rightarrow \mathbb{R}$, a *minimum spanning tree (MST)* is a spanning tree S of G such that for all spanning trees T of G , we have $w(S) \leq w(T)$.

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

Kruskal's Algorithm

<p>1 Given a network.....</p> 	<p>2 Choose the shortest edge (if there is more than one, choose any of the shortest).....</p> 	<p>3 Choose the next shortest edge and add it.....</p> 
<p>4 Choose the next shortest edge which wouldn't create a cycle and add it.</p> 	<p>5 Choose the next shortest edge which wouldn't create a cycle and add it.</p> 	<p>6 Repeat until you have a minimal spanning tree.</p> 

- └ Minimum spanning trees
 - └ Kruskal's algorithm

Algorithm 4 Kruskal's algorithm

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

procedure 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
 $U.\text{union}(u, v)$
 $S \leftarrow S \cup \{e\}$

If vertices in different components remain, the graph is not connected.

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 $E' \subseteq S$.

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.

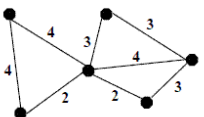


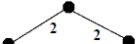
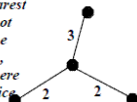
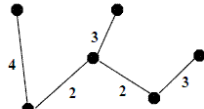
When $e \in S$, then $S' := S$ fulfills the requirement.

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 also a spanning tree, and $w(S') \leq w(S)$, as $w(e) \leq w(f)$. As S is a minimum spanning tree, we have $w(S') = w(S)$, and therefore S' is also a minimum spanning tree.

- └ Minimum spanning trees
 - └ Prim's algorithm

Kruskal and Prim

Prim's Algorithm

<p>1 Given a network.....</p> 	<p>2 Choose a vertex</p> 	<p>3 Choose the shortest edge from this vertex.</p> 
<p>4 Choose the nearest vertex not yet in the solution.</p> 	<p>5 Choose the next nearest vertex not yet in the solution, when there is a choice choose either.</p> 	<p>6 Repeat until you have a minimal spanning tree.</p> 

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 neighbors grey.
- ③ Repeat until no grey nodes are left.

- └ Minimum spanning trees
 - └ Prim's algorithm

Algorithm 5 Prim's algorithm

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

procedure PRIM(G)

$S \leftarrow \emptyset$

for each vertex $v \in V$ **do**

$visited(v) \leftarrow \text{false}$

$c(v) \leftarrow \infty$

$PQ \leftarrow \text{PriorityQueue over } V$

$s \leftarrow \text{any } v \in V$

PRIMVISIT(s)

while PQ is not empty **do**

$v \leftarrow PQ.\text{deleteMin}()$

$S \leftarrow S \cup \{\{pre(v), v\}\}$

PRIMVISIT(v)

procedure PRIMVISIT(v)

$visited(v) \leftarrow \text{true}$

for each $u \in vE$ **do**

if not $visited(u)$ **then**

if $w(v, u) < c(u)$ **then**

$pre(u) \leftarrow v$

$c(u) \leftarrow w(v, u)$

if u in PQ **then**

$PQ.\text{decreaseKey}(u, c(u))$

else

$PQ.\text{insert}(u, c(u))$

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: In Java and C++, there is no decreaseKey operation, instead delete and insert again.