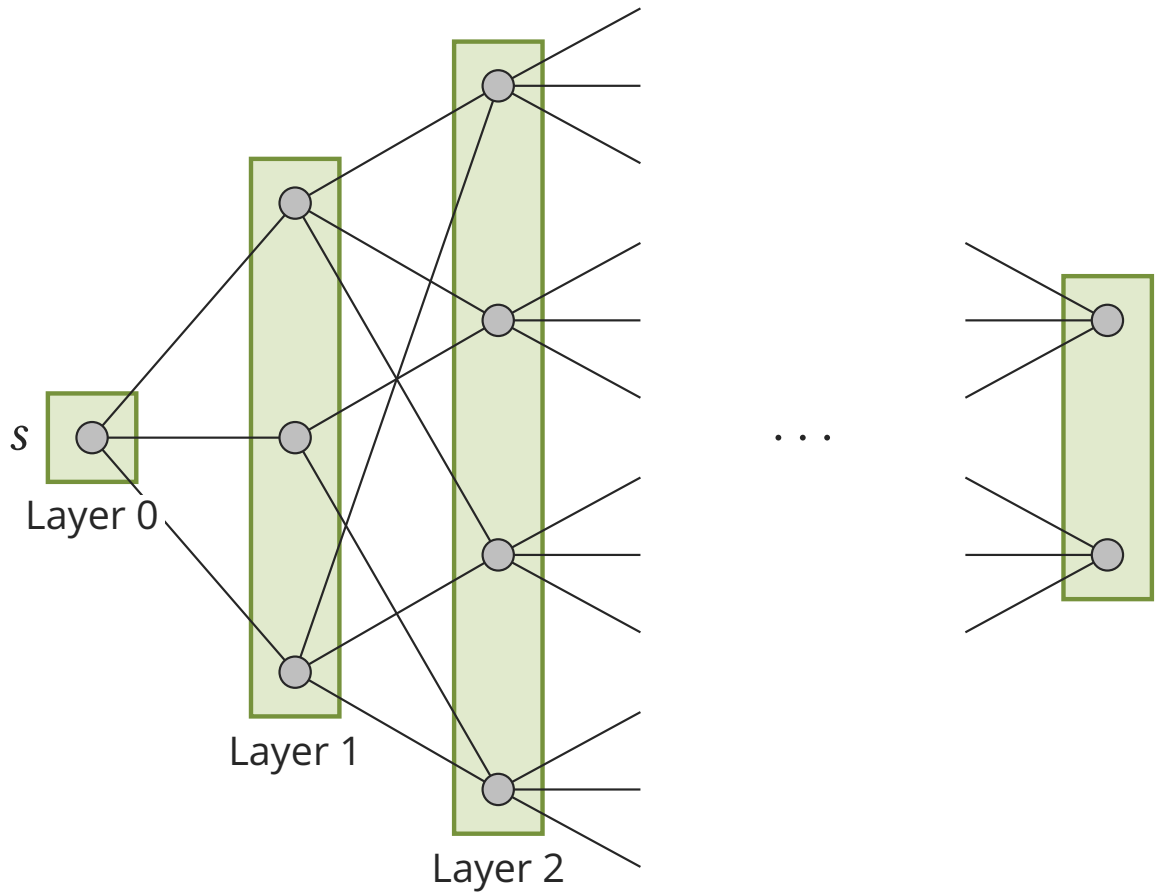
Graph Search + DAGs

Breadth First Search

Breadth First Search (BFS)

Idea

- Explore a graph layer by layer from a start vertex s.



BFS

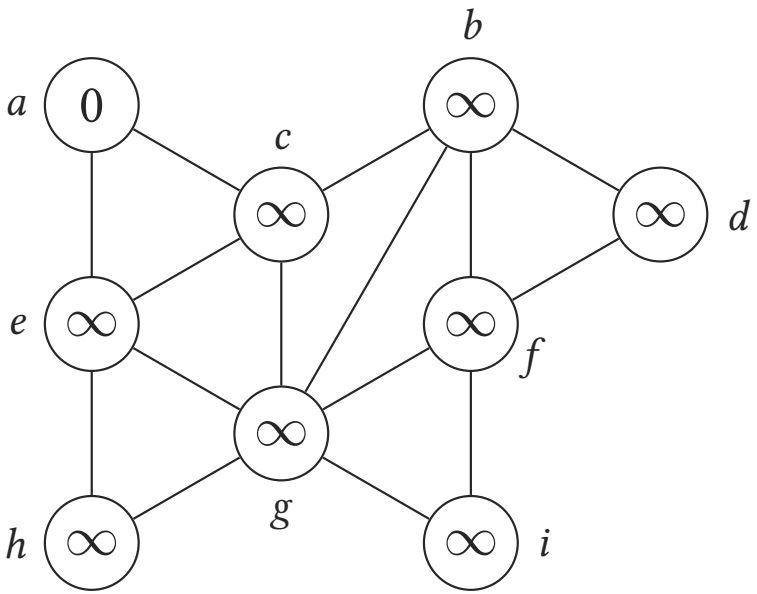
Applications

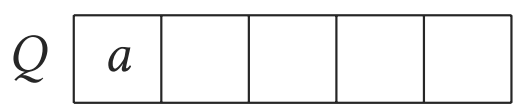
- Finding a shortest path.
- Determine distances.
- Garbage collection (checking for connected components)
- Solving Puzzles
- Web crawling
- Base for other algorithms.

BFS – Algorithm

Preparation

- For every vertex v, set the distance $\mathrm{dist}(v) := \infty$ and the parent $\mathrm{par}(v) := \texttt{null}$. For the start vertex a, set $\mathrm{dist}(a) := 0$.
- Add a to an empty queue Q.

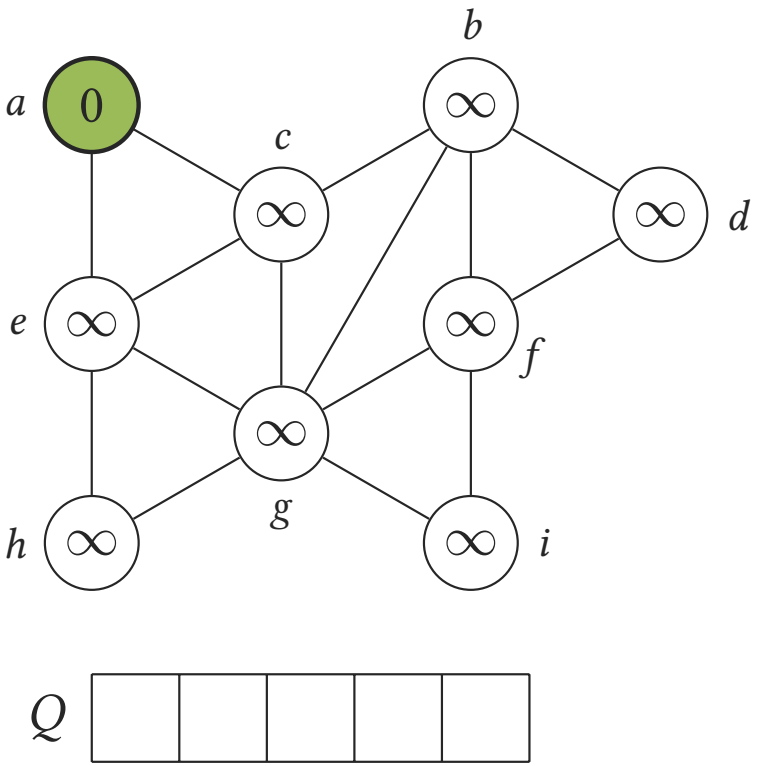




BFS – Algorithm

Iteration

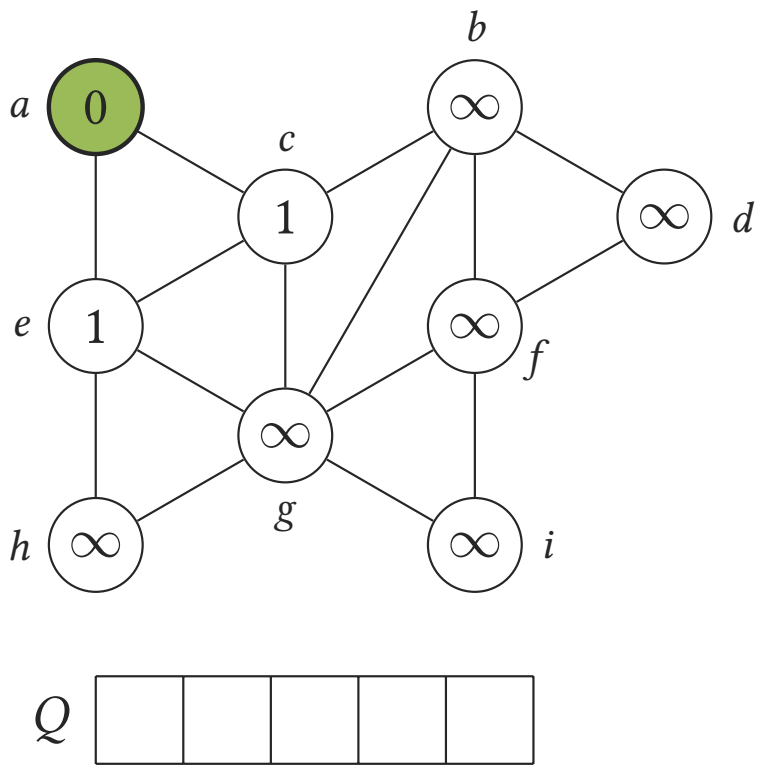
- Select and remove first vertex v from Q.



BFS – Algorithm

Iteration

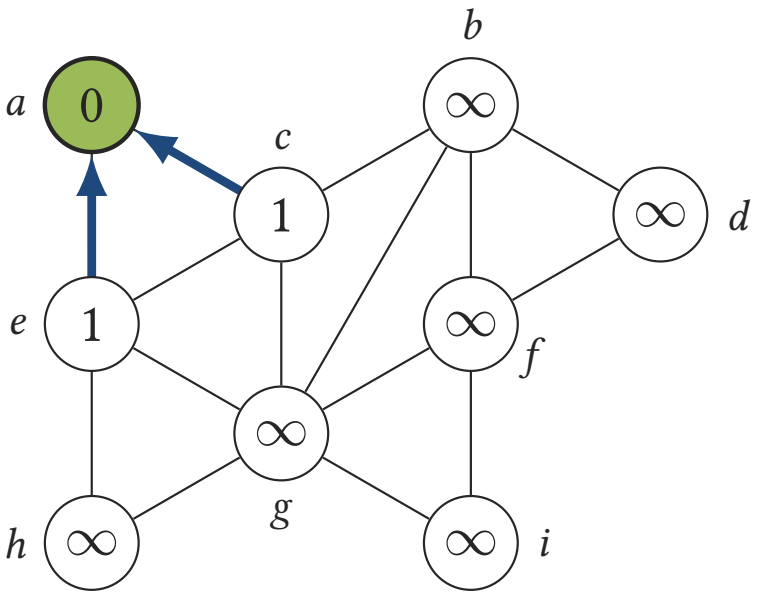
- For each $u \in N(v)$ with $\text{dist}(u) = \infty$,
  - set $\text{dist}(u) := \text{dist}(v) + 1$,



BFS – Algorithm

Iteration

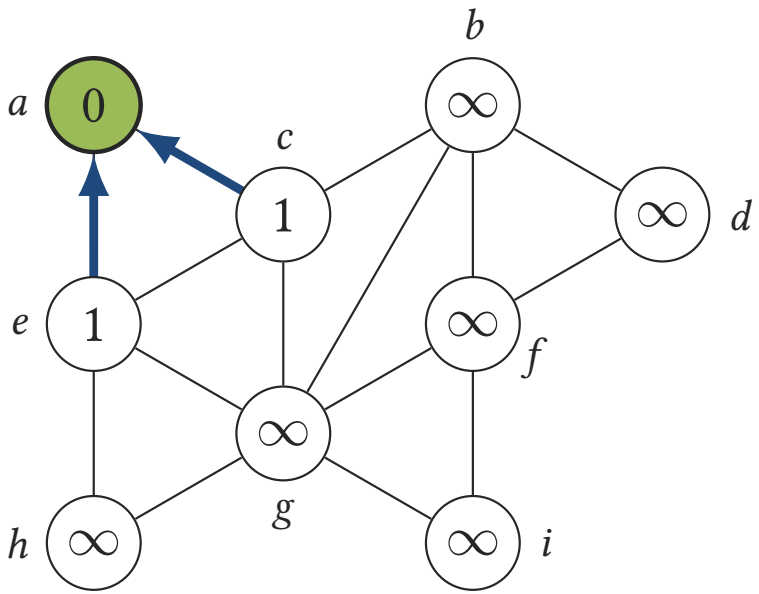
- For each $u \in N(v)$ with $\mathrm{dist}(u) = \infty$,
  - set $\mathrm{dist}(u) := \mathrm{dist}(v) + 1$,
  - set $\mathrm{par}(u) := v$, and



BFS – Algorithm

Iteration

- For each $u \in N(v)$ with $\mathrm{dist}(u) = \infty$,
  - set $\mathrm{dist}(u) := \mathrm{dist}(v) + 1$,
  - set $\mathrm{par}(u) := v$, and
  - add u to Q.

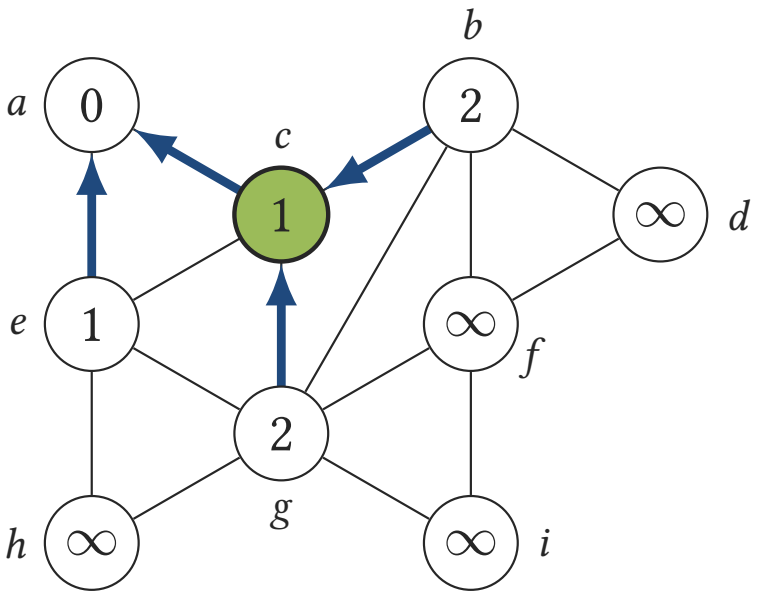


Q	c	e			

BFS – Algorithm

Iteration

- Repeat until Q is empty.

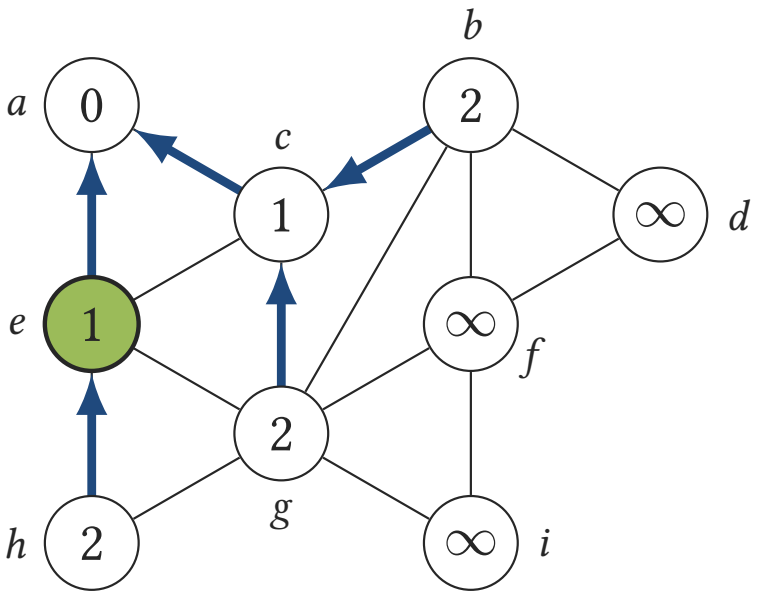


Q	e	b	g		

BFS – Algorithm

Iteration

- Repeat until Q is empty.

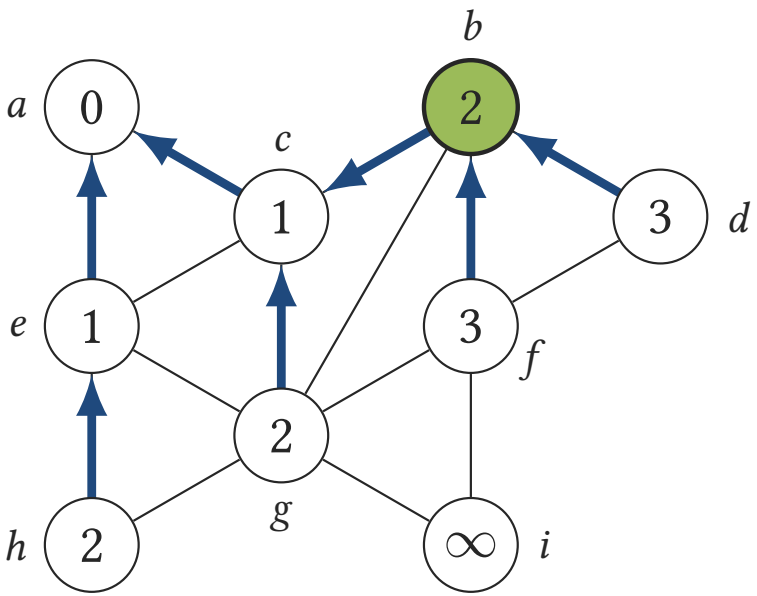


Q	b	g	h		

BFS – Algorithm

Iteration

- Repeat until Q is empty.

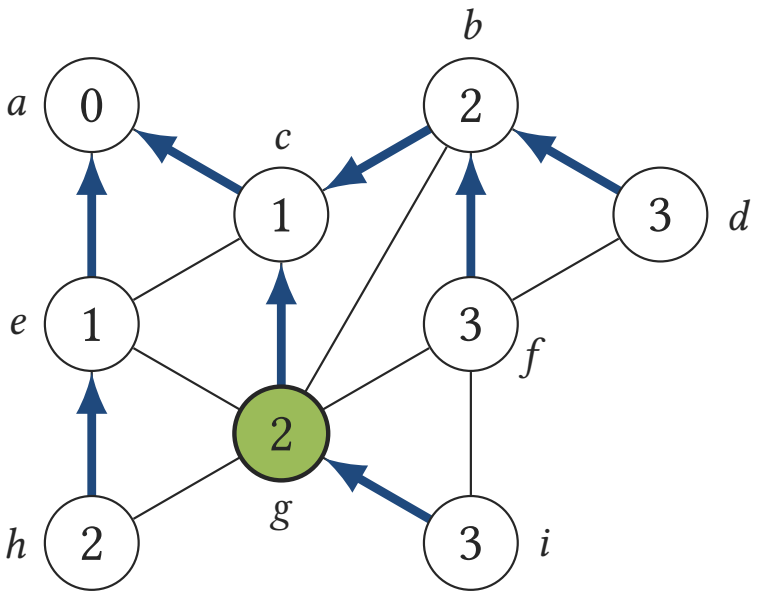


Q	g	h	d	f	

BFS – Algorithm

Iteration

- Repeat until Q is empty.



Q	h	d	f		

BFS

Input: A graph $G = (V, E)$ and a start vertex $s \in V$.

1 **For Each** $v \in V$

2 Set $\mathrm{dist}(v) := \infty$ and $\mathrm{par}(v) := \texttt{null}$.

3 Create a new empty queue Q.

4 Set $\mathrm{dist}(s) := 0$ and add s to Q.

5 **While** Q is not empty

6 $v := Q.\mathrm{deque}()$

7 **For Each** $u \in N(v)$ with $\mathrm{dist}(u) = \infty$

8 Set $\mathrm{dist}(u) := \mathrm{dist}(v) + 1$ and set $\mathrm{par}(u) = v$.

9 Add u to Q.

BFS – Runtime

Preparation

- Each vertex is accessed once. No edge is accessed.
- $\mathcal{O}(|V|)$ time

Iteration

- Each vertex is added to the queue at most once.
- For each vertex removed from the queue, each neighbour is accessed once.
- Thus, runtime is

$$\sum_{v \in V} |N(v)| = 2|E|$$

Total runtime

- $\mathcal{O}(|V| + |E|)$

Depth First Search

Depth First Search

Idea

- Follow path until you get stuck.
- If got stuck, backtrack path until reach unexplored neighbour. Continue on unexplored neighbour.

Applications

- Tree-traversal
- Cycle detection
- Mace generation
- Base for other algorithm

DFS – Algorithm (using recurrence)

Preparation

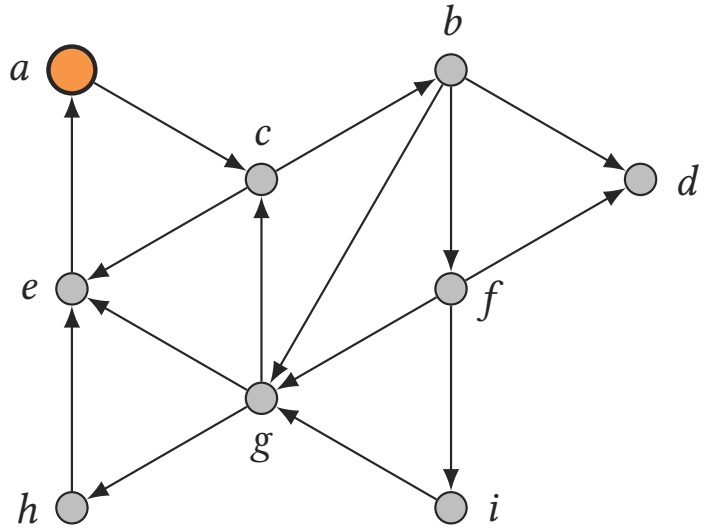
- For every vertex v, set it as unvisited ($\mathrm{vis}(v) := \mathrm{False}$) and set the parent $\mathrm{par}(v) := \texttt{null}$.
- Call DFS(s) for the start vertex s.

```
1 Procedure DFS(v)
2     Set vis(v) = True.
3     For Each u ∈ N(v) with vis(u) = False
4         Set par(u) := v.
5         Call DFS(u).
```

(For large graphs, do not use a recursive implementation.)

DFS – Algorithm Example

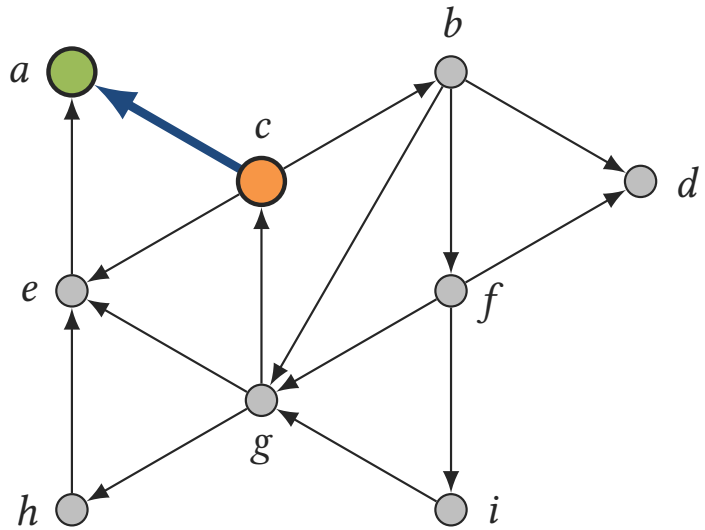
Run a DFS with start vertex s.



Stack:

DFS – Algorithm Example

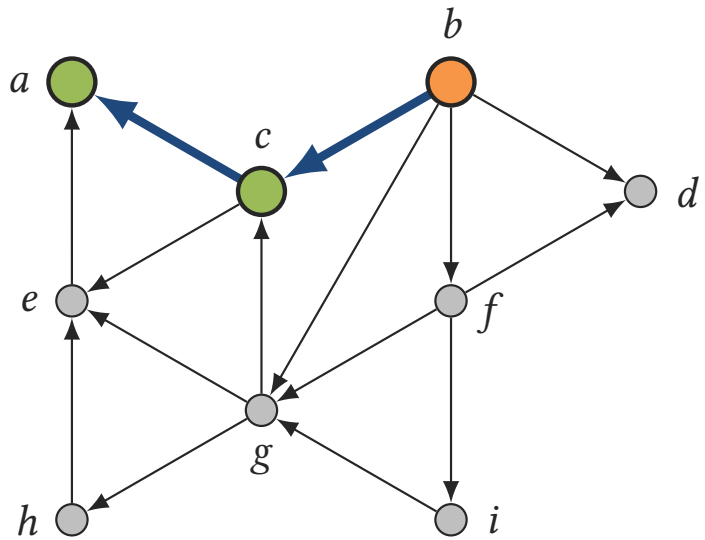
Run a DFS with start vertex s.



Stack: a

DFS – Algorithm Example

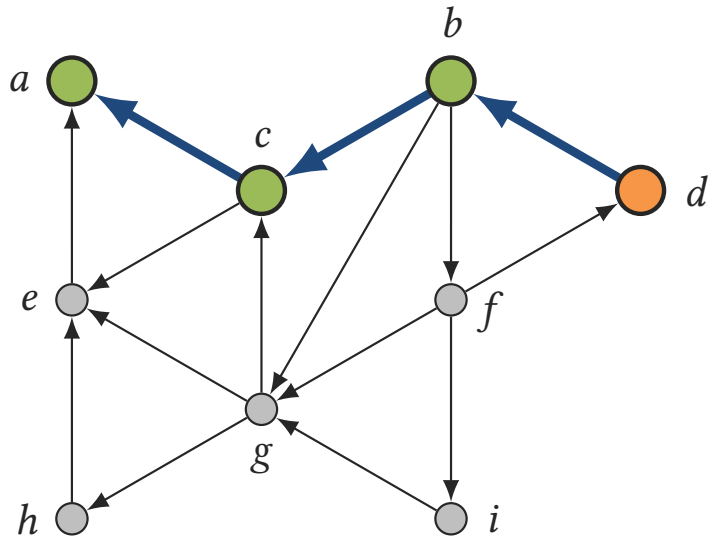
Run a DFS with start vertex s.



Stack: a c

DFS – Algorithm Example

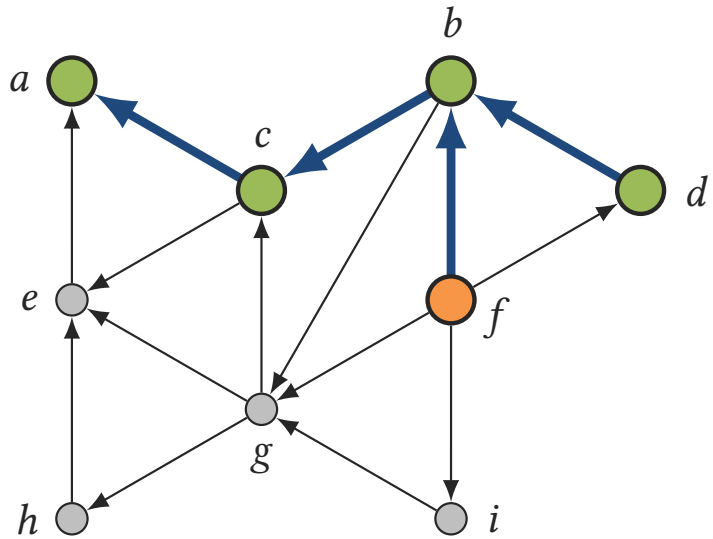
Run a DFS with start vertex s.



Stack: a c b

DFS – Algorithm Example

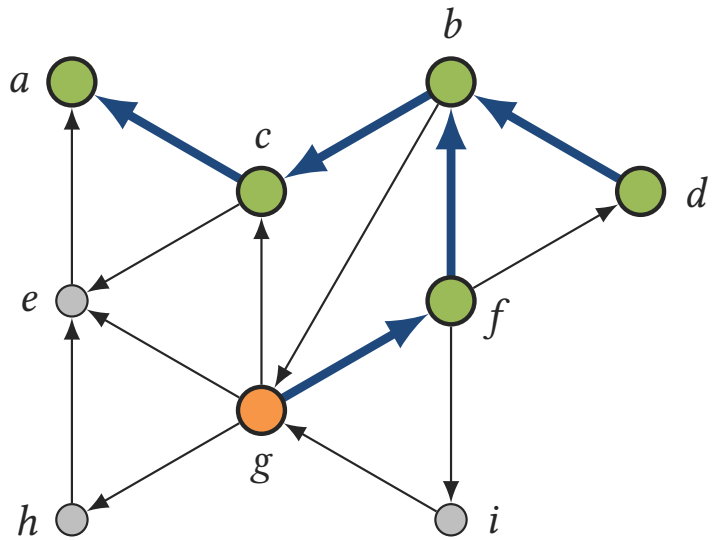
Run a DFS with start vertex s.



Stack: a c b

DFS – Algorithm Example

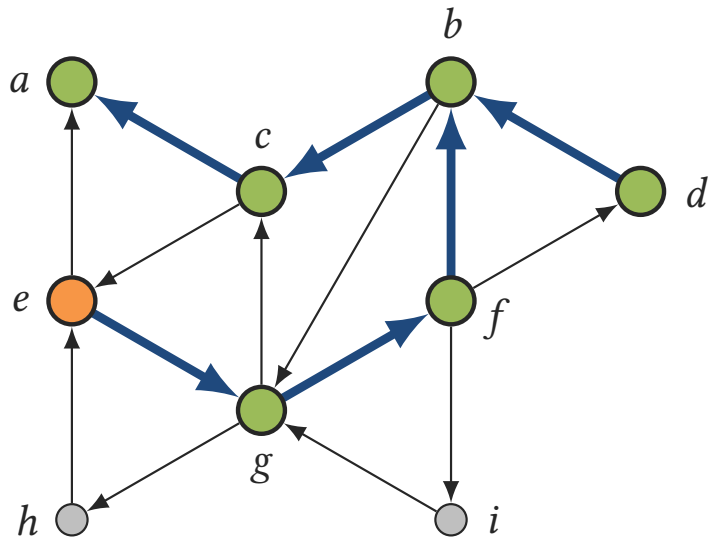
Run a DFS with start vertex s.



Stack: a c b f

DFS – Algorithm Example

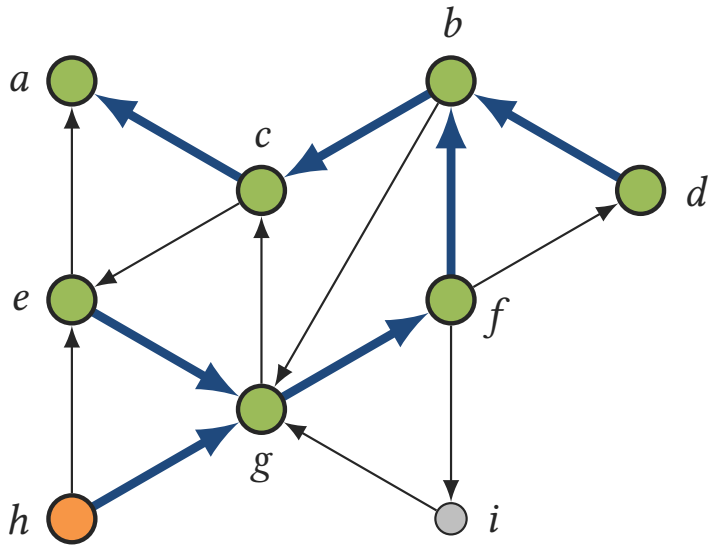
Run a DFS with start vertex s.



Stack: a c b f g

DFS – Algorithm Example

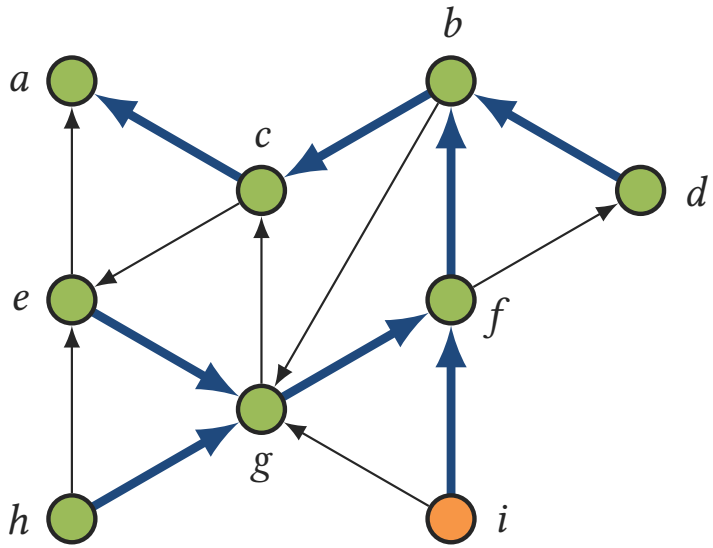
Run a DFS with start vertex s.



Stack: a c b f g

DFS – Algorithm Example

Run a DFS with start vertex s.



Stack: a c b f

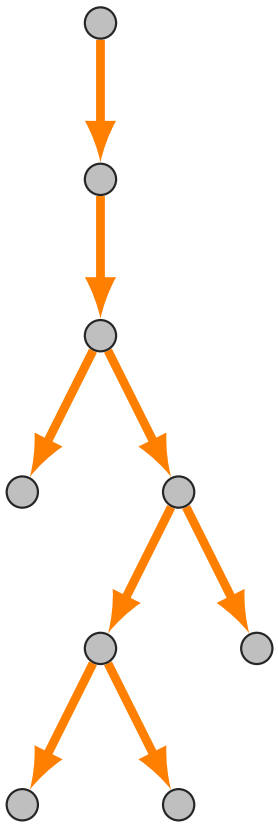
DFS – Runtime

Runtime

- A single DFS(v) call (without recurrence) accesses v and all its neighbours. Thus, runtime is $\mathcal{O}(|N[v]|)$ for a single vertex v.
- For each vertex v, DFS(v) is called only once.
- Total runtime: $\mathcal{O}(|V| + |E|)$

DFS – Edges

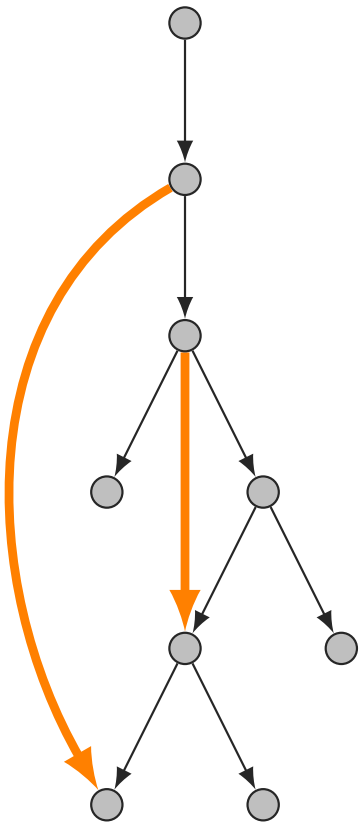
A DFS partitions the edges in four groups.



1. **Tree edges.**
   Determined by parent pointers $\mathrm{par}(\cdot)$.

DFS – Edges

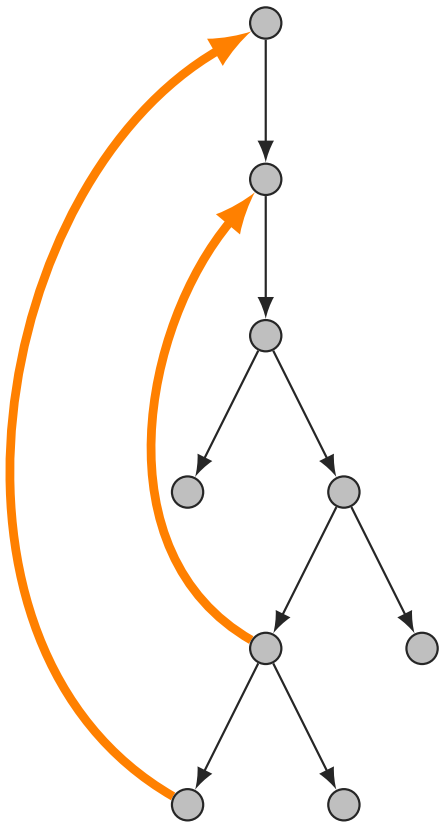
A DFS partitions the edges in four groups.



1. **Tree edges.**
   Determined by parent pointers $\mathrm{par}(\cdot)$.
2. **Forward edges.**
   From an ancestor to a descented

DFS – Edges

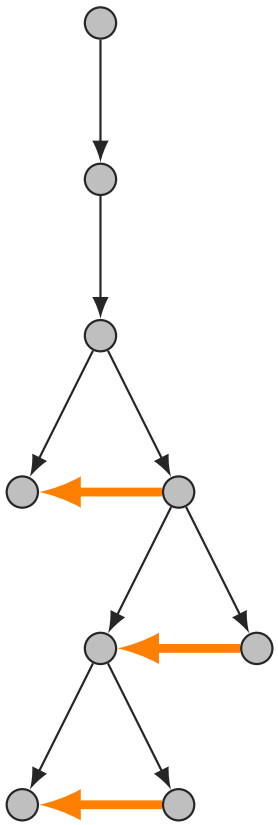
A DFS partitions the edges in four groups.



1. **Tree edges.**
   Determined by parent pointers $\mathrm{par}(\cdot)$.
2. **Forward edges.**
   From an ancestor to a descented
3. **Back edges.**
   From a descented to an ancestor

DFS – Edges

A DFS partitions the edges in four groups.



1. **Tree edges.**
   Determined by parent pointers $\mathrm{par}(\cdot)$.
2. **Forward edges.**
   From an ancestor to a descented
3. **Back edges.**
   From a descented to an ancestor
4. **Cross edges.** (only in directed graphs)
   Remaining edges

DFS – Recognising Edges

On the DFS-tree

- Make a preorder and a postorder traversal.
- For each vertex, store a vector (i, j) where i is the index of v in the preorder and j is the index of v in the postorder.

For an edge uv

- Let (i, j) be the indices for u and (x, y) be the indices for v.
- Forward edge: $i < x$
- Backward edge: $i > x$ and $j < y$
- Cross edge: $i > x$ and $j > y$

DFS – Detecting Cycles

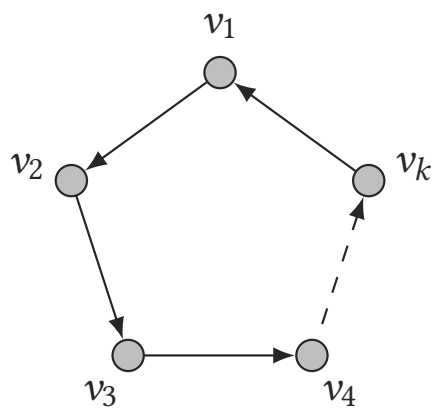
Theorem

A graph G has a cycle if and only if any DFS has a back edge.

DFS – Detecting Cycles

Proof ($\Rightarrow$)

- Assume G has a cycle $\{v_1, v_2, \ldots, v_k\}$. W. l. o. g., let v_1 be the first visited by the DFS.

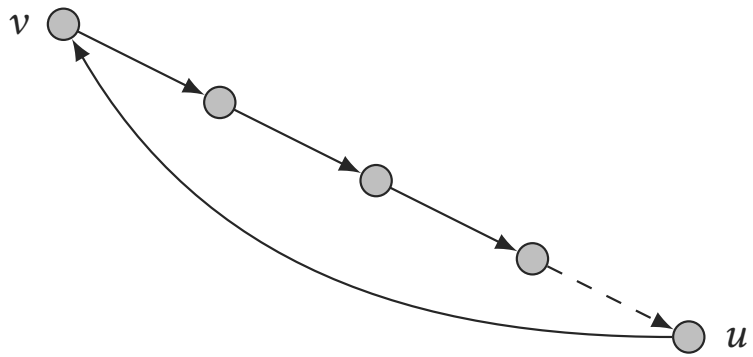


- Then, v_k is an descendent of v_1 in the DFS tree, i. e., $v_k v_1$ is a back edge. □

DFS – Detecting Cycles

Proof ($\Leftarrow$)

- Assume a DFS produces a back edge uv.



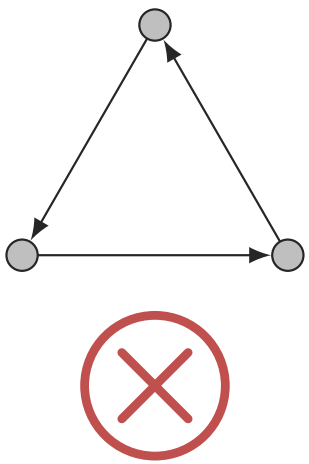
- Thus, v is an ancestor of u, i. e., the path from v to u using tree edges plus the edge uv form a cycle. □

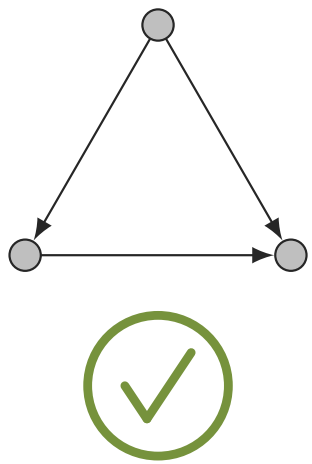
Directed Acyclic Graphs

Directed Acyclic Graphs

Directed Acyclic Graphs (DAG)

A *directed acyclic graph* (or DAG for short) is a directed graph that contains no cycles.





Directed Acyclic Graphs

Cycles + DFS

- A graph contains a cycle if and only if a DFS produces a back edge.
- Thus, if a graph is acyclic, a DFS on this graph produces no back edges.

Lemma

Determining if a given directed graph is a DAG can be done in linear time.

Partial Orders

- Relation which is transitive, reflexive, and antisymmetric.
- For each partial order there is a corresponding DAG and vice versa.

Sources and Sinks

Source and Sink

In a DAG, a *source* is a vertex without incoming edges; a *sink* is a vertex without outgoing edges.

Sources and Sinks

Lemma

Each DAG contains at least one source and one sink.

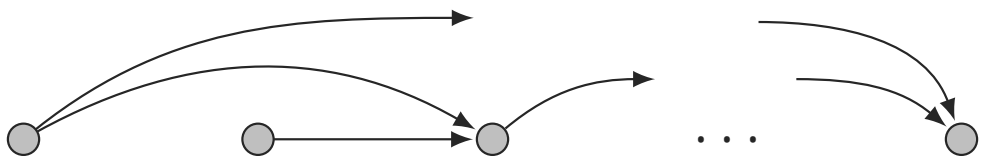
Proof

- Pick arbitrary vertex v.
- If there is a vertex u with $(v, u) \in E$, go to u.
- Repeat this for u.
- Since V is finite and there are no cycles, u will be a sink eventually.
- By symmetry, the same for sources. □

Topological Order

Topological Order

For a directed graph, a vertex order $\langle v_1, v_2, \ldots, v_n \rangle$ ($v_i \neq v_j \leftrightarrow i \neq j$) is a *topological order* if $(v_i, v_j) \in E$ implies $i < j$.



Topological Order

Lemma

A graph is a DAG if and only if admits a topological order.

Proof ($\Leftarrow$)

- If a graph admits a topological order, it cannot contain cycles. □

Proof ($\Rightarrow$)

- After removing or adding a source or a sink from or to a DAG, the resulting graph is still a DAG.
- The first vertex of a topological order is a source, the last is a sink.
- Thus, by induction, each DAG admits a topological order. □

Finding a Topological Order

- A post-order on a DFS-tree gives a topological order.

Exercises

Exercises

In an undirected graph $G = (V, E)$, the *eccentricity* $\mathrm{ecc}(v)$ of a vertex v, the *diameter* $\mathrm{diam}(G)$, and the *radius* $\mathrm{rad}(G)$ of G are defined as follows:

- $\mathrm{ecc}(v) = \max_{u \in V} d(u, v)$
- $\mathrm{diam}(G) = \max_{v \in V} \mathrm{ecc}(v)$
- $\mathrm{rad}(G) = \min_{v \in V} \mathrm{ecc}(v)$

Give an algorithm that determines the diameter and radius of a given graph. State the runtime of your algorithm.

Exercises

Let $G = (V, E)$ be an undirected graph. For a vertex $v \in V$ and a set $S \subseteq V$, the *projection* $\mathrm{Pr}(v, S)$ is the set of vertices in S with minimal distance to v. Formally, $\mathrm{Pr}(v, S) = \{\, u \in S \mid d(u, v) = d(v, S) \,\}$.

Give an algorithm that determines the *projection* $\mathrm{Pr}(v, S)$ in linear time for a graph G, vertex v and vertex set S.

Exercises

In a DAG G, an edge (u, v) is *transitive* if, after removing it, there is still a path from u to v. Give an algorithm that determines all transitive edges in a given DAG.

Give an $\mathcal{O}(|V| + |E|)$ time algorithm to remove all the cycles in a directed graph $G = (V, E)$. Removing a cycle means removing an edge of the cycle. If there are k cycles in G, the algorithm should only remove at most $\mathcal{O}(k)$ edges.

Give an algorithm that determines if a given undirected graph is a tree in $\mathcal{O}(|V|)$ time, i. e., the runtime should not depend on the number of edges.

Exercises

You are given an undirected graph G and an integer k. Give a linear time algorithm that finds the maximum induced subgraph H of G such that each vertex in H has degree at least k, or proves that no such graph exists.

Consider a graph $G = (V, E)$ where $V = \{1, \ldots, n\}$. We say that an adjacency list representation of G is *monotone* if, for every vertex v, the vertices adjacent to i are listed in increasing order. Given an adjacency list representation of G, produce a monotone adjacency list representation of G in linear time.