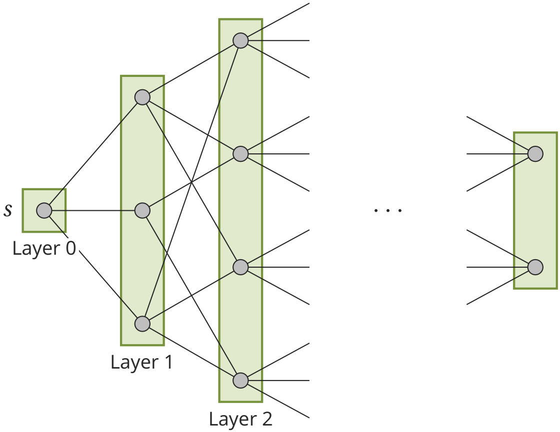
Graph Search

Breadth First Search

Breadth First Search (BFS)

Idea

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



BFS

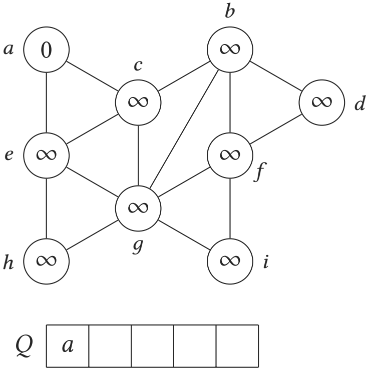
Applications

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

BFS – Algorithm

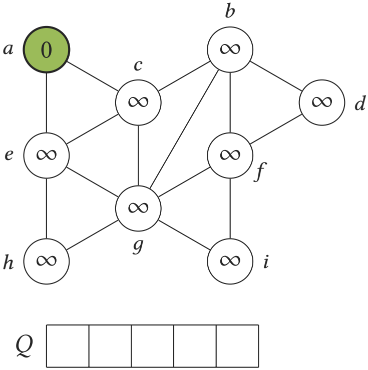
Preparation

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



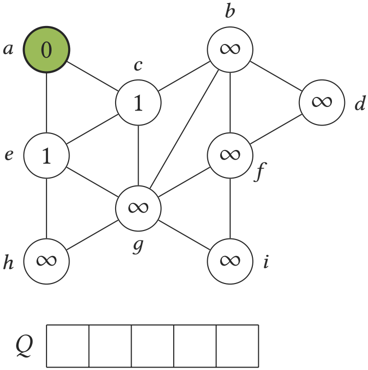
Iteration

- Select and remove first vertex v from Q.



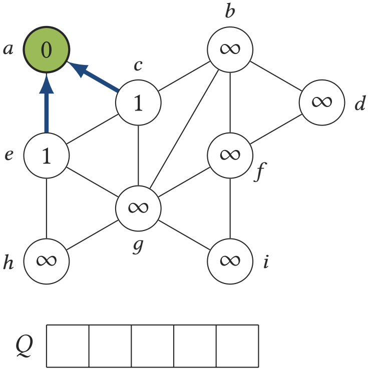
Iteration

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



Iteration

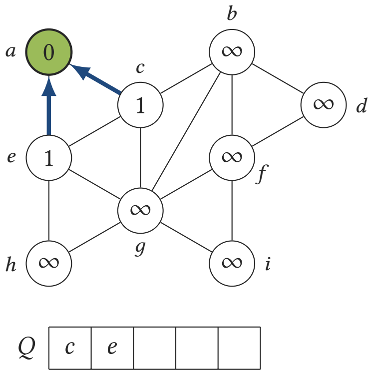
- For each $u \in N(v)$ with $\mathrm{dist}(u) = \infty$,
  - set $\mathrm{dist}(u) := \mathrm{dist}(v) + 1$,
  - set $\mathrm{par}(u) := v$, and



BFS – Algorithm

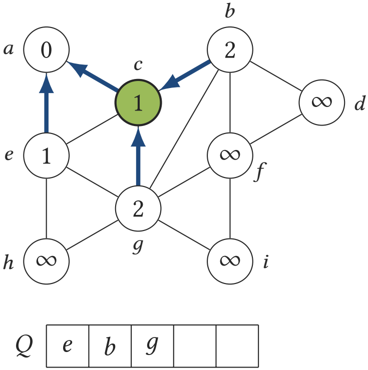
Iteration

- 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.



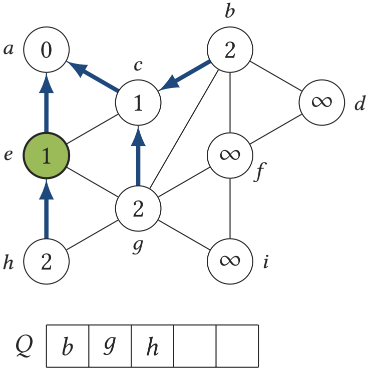
Iteration

- Repeat until Q is empty.



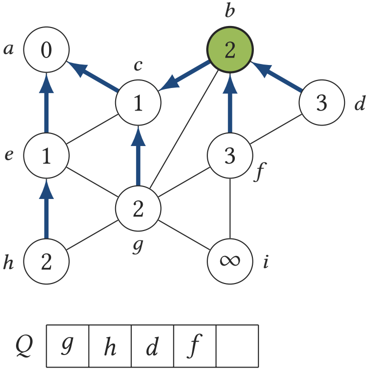
Iteration

- Repeat until Q is empty.



Iteration

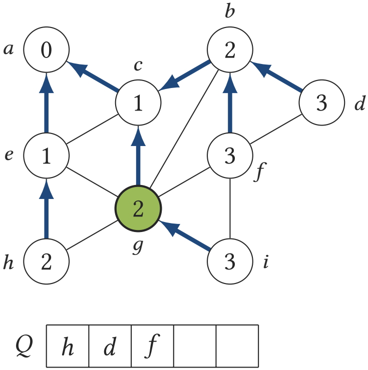
- Repeat until Q is empty.



BFS – Algorithm

Iteration

- Repeat until Q is empty.



BFS

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

For Each $v \in V$
- Set $\mathrm{dist}(v) := \infty$ and $\mathrm{par}(v) := \texttt{null}$.

Create a new empty queue Q.

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

While Q is not empty
- $v := Q$.deque()
- **For Each** $u \in N(v)$ with $\mathrm{dist}(u) = \infty$
  - Set $\mathrm{dist}(u) := \mathrm{dist}(v) + 1$ and set $\mathrm{par}(u) = v$.
  - 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

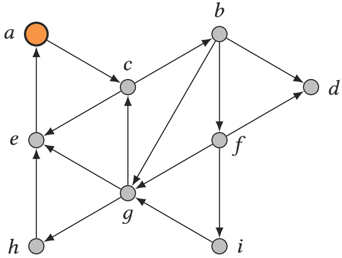
- For every vertex v, set it as unvisited ($\mathrm{vis}(v) := \mathrm{False}$) and set the parent $\mathrm{par}(v) := \mathtt{null}$.
- Call DFS(s) for the start vertex s.

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

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

DFS – Algorithm Example

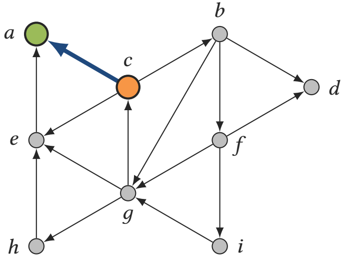
Run a DFS with start vertex s.



Stack:

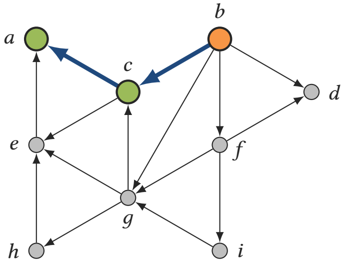
DFS – Algorithm Example

Run a DFS with start vertex s.



Stack: a

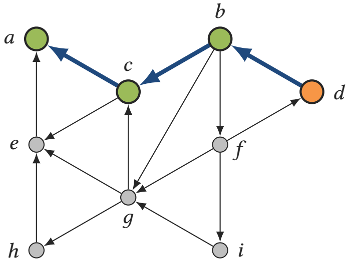
Run a DFS with start vertex s.



Stack: a c

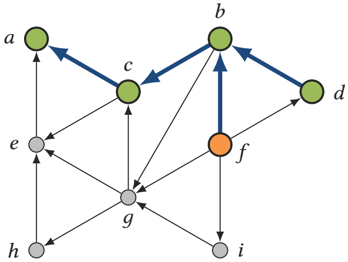
DFS – Algorithm Example

Run a DFS with start vertex s.



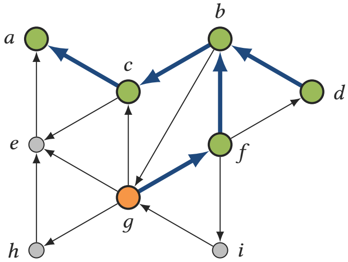
Stack: a c b

Run a DFS with start vertex s.



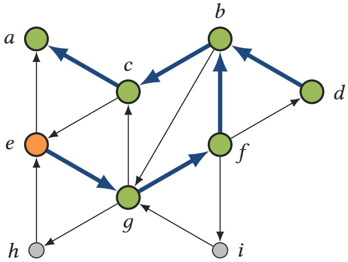
Stack: a c b

Run a DFS with start vertex s.



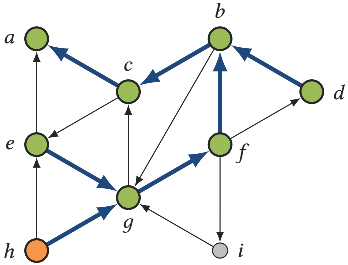
Stack: a c b f

Run a DFS with start vertex s.



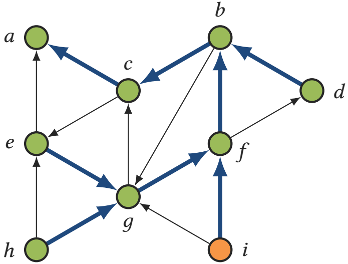
Stack: a c b f g

Run a DFS with start vertex s.



Stack: a c b f g

Run a DFS with start vertex s.



Stack: a c b f

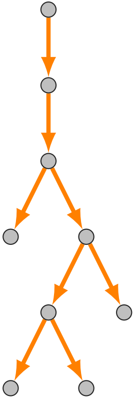
DFS – Runtime

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

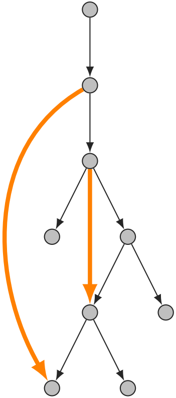
A DFS partitions the edges in four groups.



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

DFS – Edges

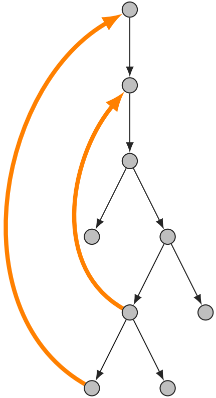
A DFS partitions the edges in four groups.



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

DFS – Edges

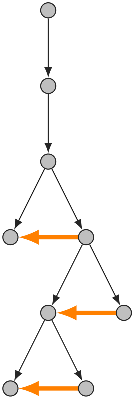
A DFS partitions the edges in four groups.



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

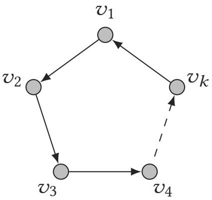
Theorem

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.

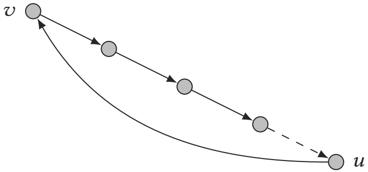


- Then, v_k is an descendent of v_1 in the DFS tree, i. e., $v_k v_1$ is a back edge. $\square$

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. $\square$