COL 351 : Analysis & Design of Algorithms

Lecture 10

Graph Algorithms III :

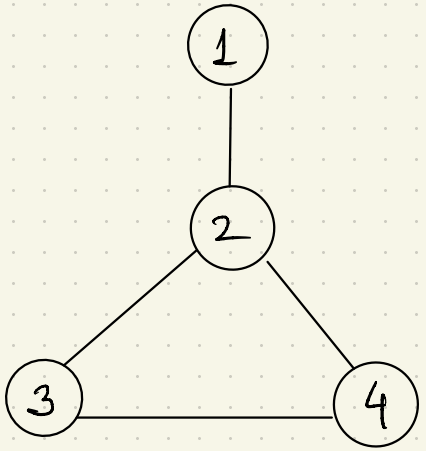
BFS, DFS, and Applications

Aug 14, 2024 | Rohit Vaish

REPRESENTING GRAPHS

Adjacency matrix

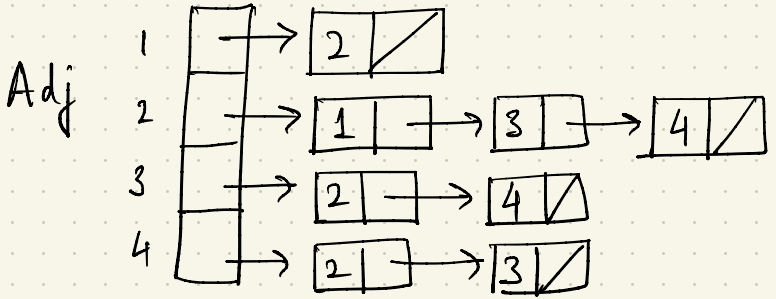
$$A = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{bmatrix} \end{array}$$

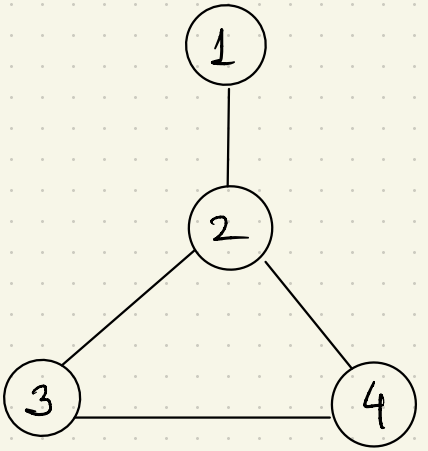


$\Theta(n^2)$ space

REPRESENTING GRAPHS

Adjacency list



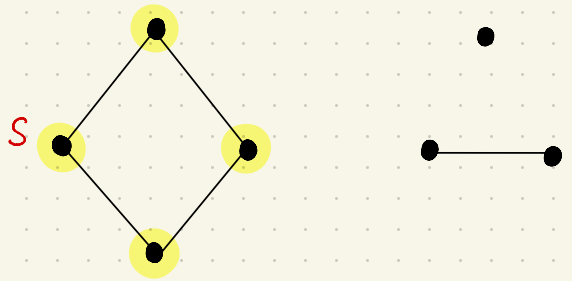


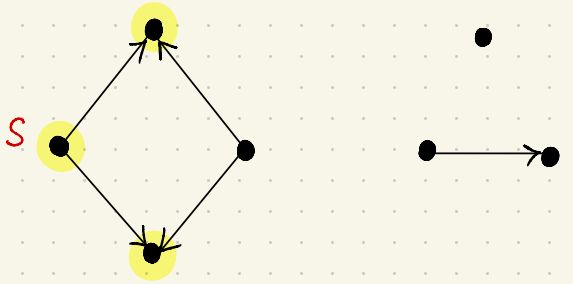
$\Theta(m+n)$ space

Generic Graph Search

input: an undirected or directed graph $G=(V,E)$, starting vertex s

goal: identify the vertices of V reachable from s





GENERIC GRAPH SEARCH

input: an undirected or directed graph $G=(V,E)$, starting vertex s

goal: identify the vertices of V reachable from s efficiently

$O(|V|+|E|)$ time

GENERIC GRAPH SEARCH

input: an undirected or directed graph $G=(V,E)$, starting vertex s

goal: identify the vertices of V reachable from s efficiently

Generic Algorithm

initially s is explored, other vertices unexplored

while some edge $(u,v) \in E$ with u explored
and v unexplored

  mark v explored

Generic Graph Search

input: an undirected or directed graph $G=(V,E)$, starting vertex s

goal: identify the vertices of V reachable from s efficiently

Generic Algorithm

initially s is explored, other vertices unexplored

while some edge $(u,v) \in E$ with u explored and v unexplored

    mark v explored

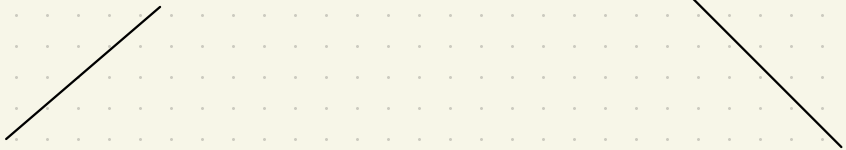
Claim: v is marked explored $\iff$ G has a path from s to v

Generic Algorithm

initially s is explored, other vertices unexplored

while some edge $(u,v) \in E$ with u explored and v unexplored

  mark v explored



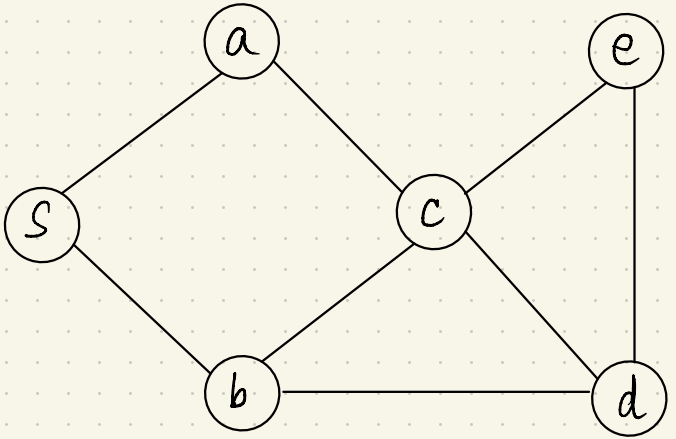
Breadth-first search (BFS)

explore layerwise

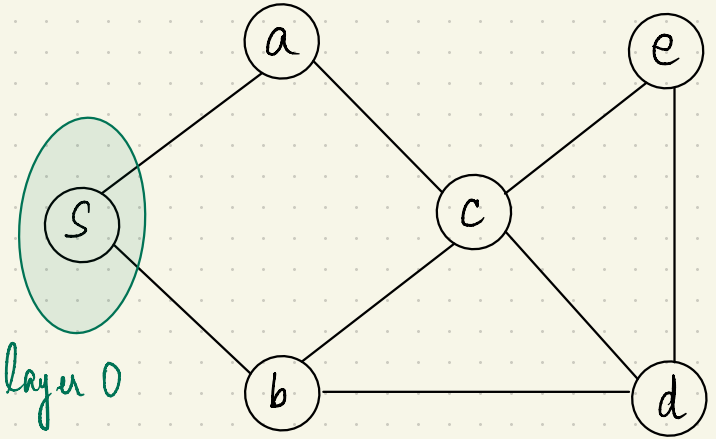
Depth-first search (DFS)

explore like a maze

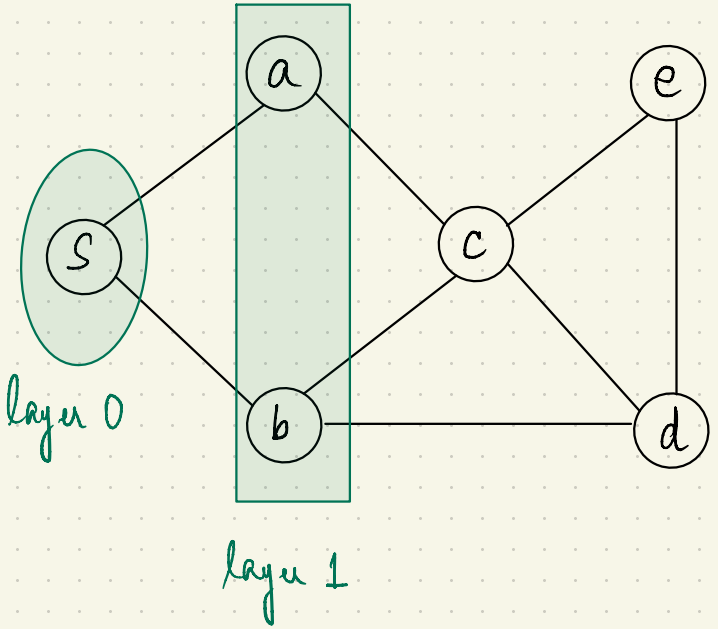
BREADTH - FIRST SEARCH



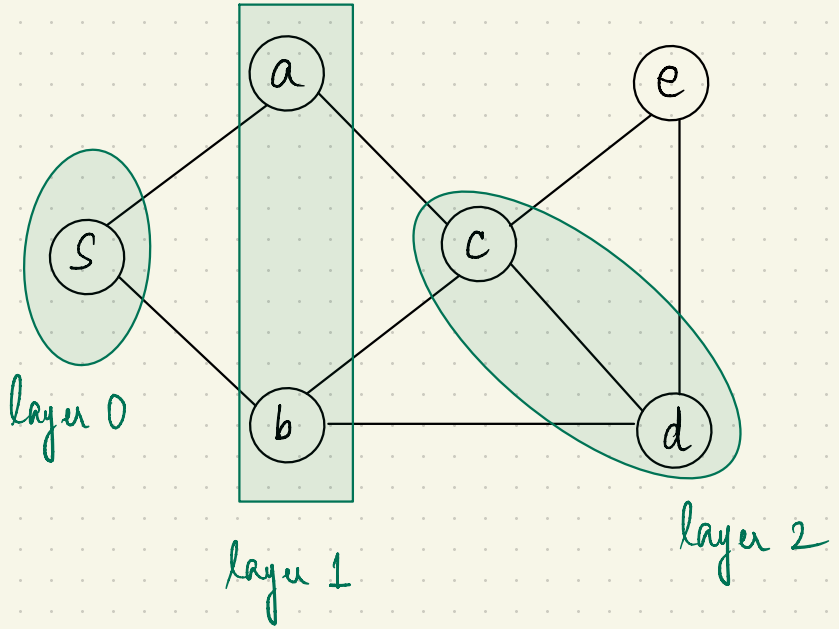
BREADTH - FIRST SEARCH



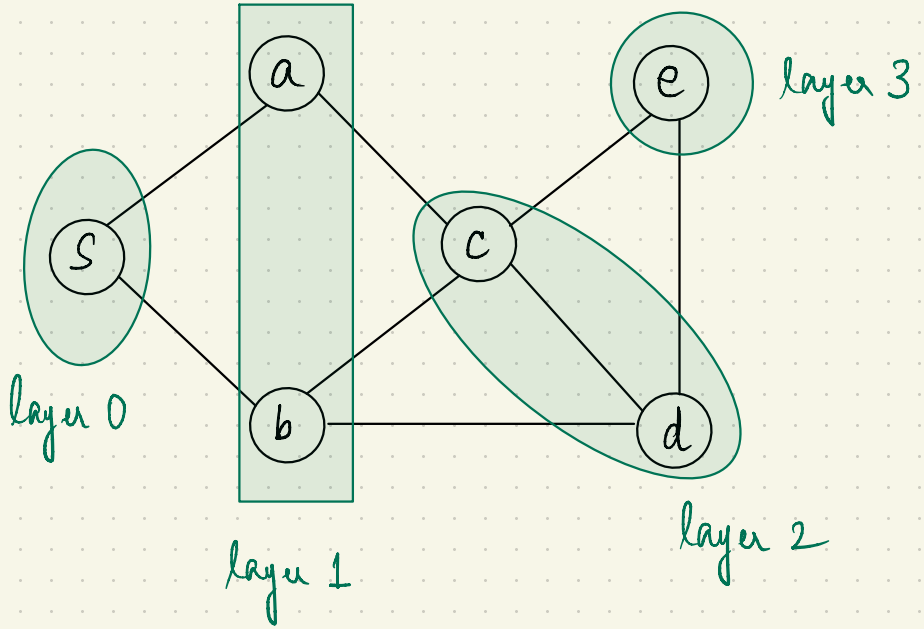
BREADTH - FIRST SEARCH



BREADTH - FIRST SEARCH



BREADTH - FIRST SEARCH



BREADTH-FIRST SEARCH

mark s as explored, all other vertices unexplored

Q := a queue data structure (FIFO), initialized with s

while $Q \neq \emptyset$

    remove the first node of Q, say v

    for each $(v, w) \in E$

        if w is unexplored

            mark w as explored

            add w to the end of Q

BREADTH-FIRST SEARCH

Claim 1: At the end of BFS

v explored $\iff$ there is a path from s to v

Claim 2: Running time of while-loop of BFS is $O(n_s + m_s)$

where n_s = # vertices reachable from s

m_s = # edges " " "

BREADTH-FIRST SEARCH

mark s as explored, all other vertices unexplored

Q := a queue data structure (FIFO), initialized with s

```
while Q ≠ ∅
    remove the first node of Q, say v
    for each (v, w) ∈ E
        if w is unexplored
            mark w as explored
            add w to the end of Q
```

Applications of BFS

Shortest paths

Connected Components

Applications of BFS

Shortest paths

Connected Components

SHORTEST PATHS

input: an undirected / directed graph $G=(V,E)$, a starting vertex s

output: $dist(s,v)$ for every vertex $v \in V$

SHORTEST PATHS

input: an undirected / directed graph $G=(V,E)$, a starting vertex s

output: $dist(s,v)$ for every vertex $v \in V$

number of edges

$$dist(s,v) = \begin{cases} \text{length of shortest path from } s \text{ to } v \text{ if one exists,} \\ \infty \quad \text{otherwise} \end{cases}$$

SHORTEST PATHS

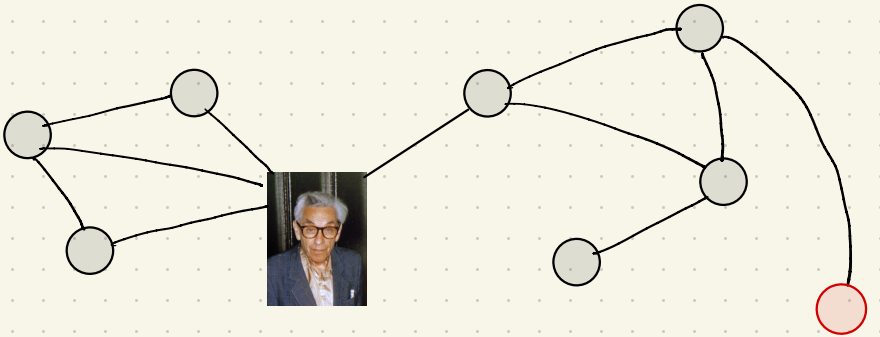
input: an undirected / directed graph $G=(V,E)$, a starting vertex s

output: $dist(s,v)$ for every vertex $v \in V$

number of edges

$dist(s,v)$ = length of shortest path from s to v if one exists,

∞ otherwise



SHORTEST PATHS

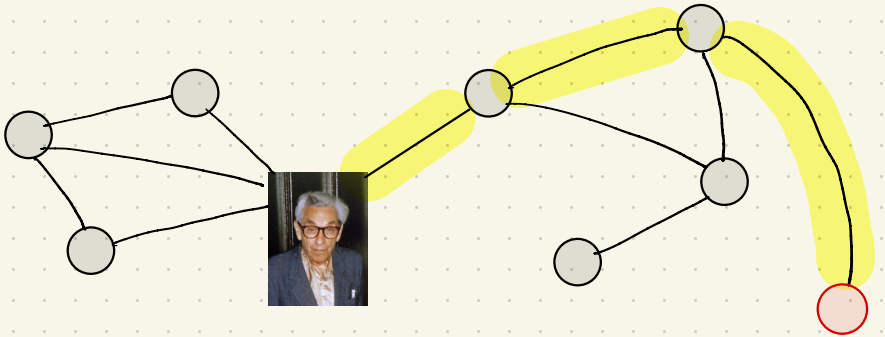
input: an undirected / directed graph $G=(V,E)$, a starting vertex s

output: $dist(s,v)$ for every vertex $v \in V$

number of edges

$dist(s,v)$ = length of shortest path from s to v if one exists,

∞ otherwise



SHORTEST PATHS

BFS pseudo code

mark s as explored, all other vertices unexplored

Q := a queue data structure (FIFO), initialized with s

while $Q \neq \emptyset$

    remove the first node of Q, say v

    for each $(v, w) \in E$

        if w is unexplored

            mark w as explored

            add w to the end of Q

SHORTEST PATHS

mark s as explored, all other vertices unexplored

Q := a queue data structure (FIFO), initialized with s

while $Q \neq \emptyset$

    remove the first node of Q, say v

    for each $(v, w) \in E$

        if w is unexplored

            mark w as explored

            add w to the end of Q

SHORTEST PATHS

mark s as explored, all other vertices unexplored

$dist(s) := 0$, $dist(v) = \infty$ for all $v \neq s$.

$Q :=$ a queue data structure (FIFO), initialized with s

while $Q \neq \emptyset$

    remove the first node of Q, say v

    for each $(v, w) \in E$

        if w is unexplored

            mark w as explored

            add w to the end of Q

SHORTEST PATHS

mark s as explored, all other vertices unexplored

$dist(s) := 0$, $dist(v) = \infty$ for all $v \neq s$.

$Q :=$ a queue data structure (FIFO), initialized with s

while $Q \neq \emptyset$

    remove the first node of Q, say v

    for each $(v, w) \in E$

        if w is unexplored

            mark w as explored

            add w to the end of Q

SHORTEST PATHS

mark s as explored, all other vertices unexplored

$dist(s) := 0$, $dist(v) = \infty$ for all $v \neq s$.

$Q :=$ a queue data structure (FIFO), initialized with s

while $Q \neq \emptyset$

    remove the first node of Q, say v

    for each $(v, w) \in E$

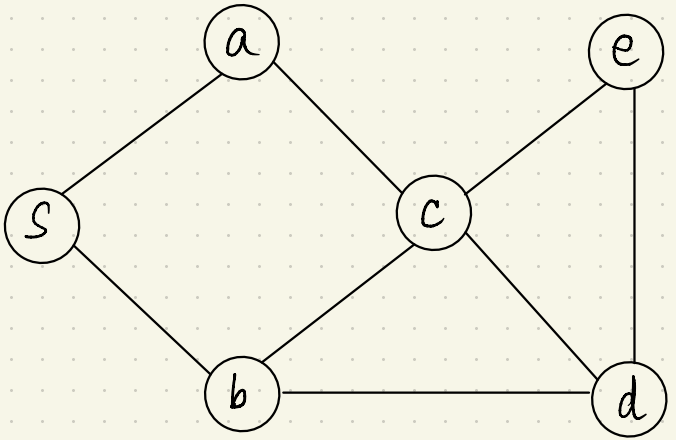
        if w is unexplored

            mark w as explored

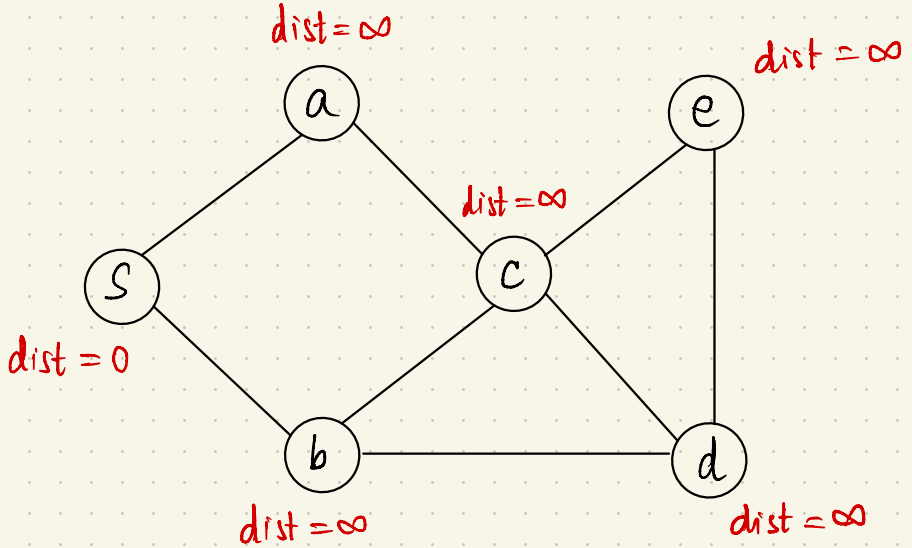
            $dist(w) := dist(v) + 1$

            add w to the end of Q

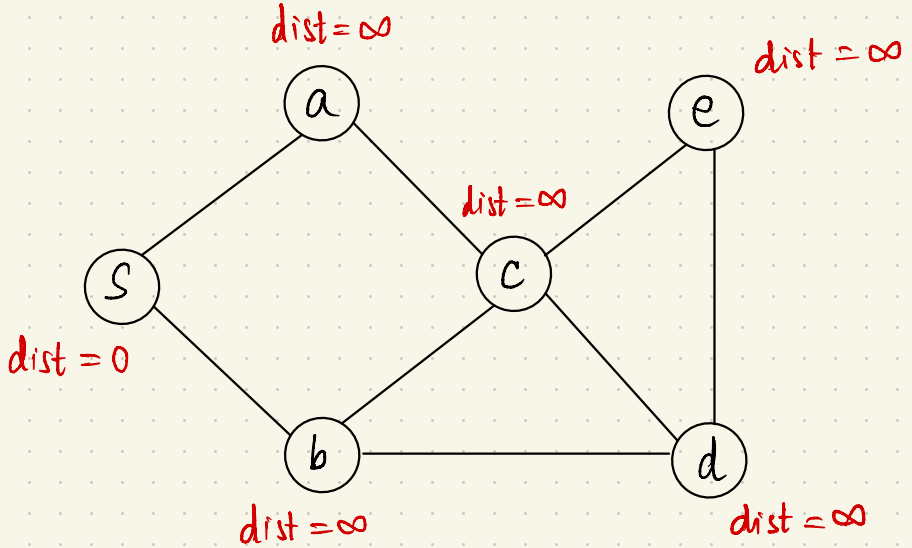
Shortest Paths



SHORTEST PATHS

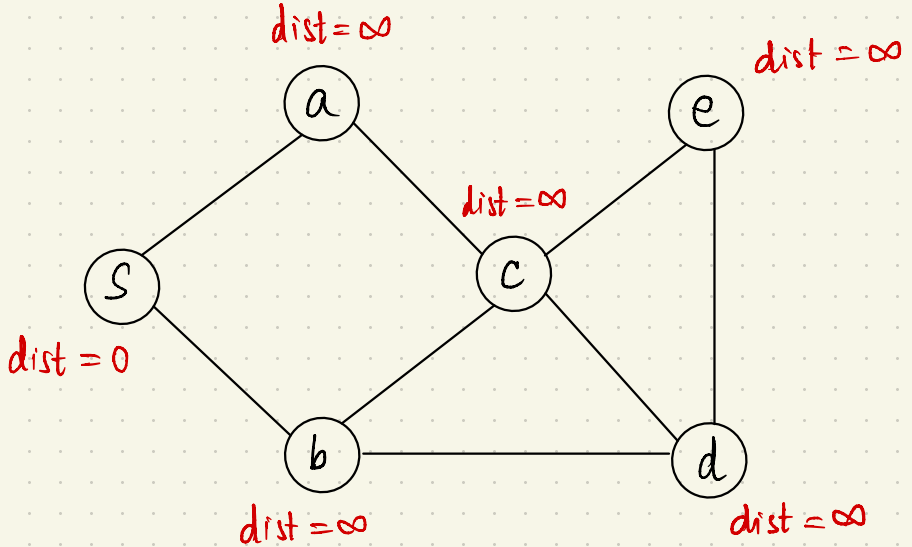


SHORTEST PATHS



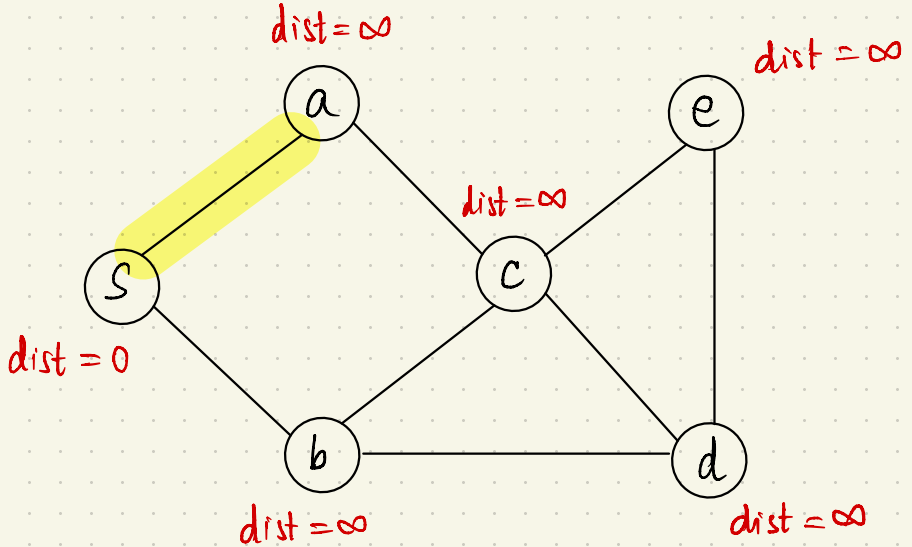
$Q = S$

SHORTEST PATHS



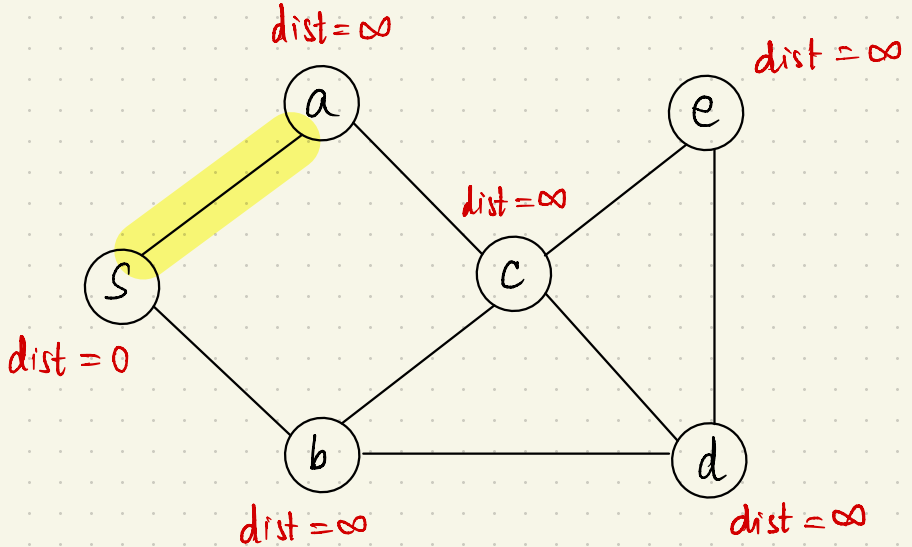
$Q = \{S\}$

SHORTEST PATHS



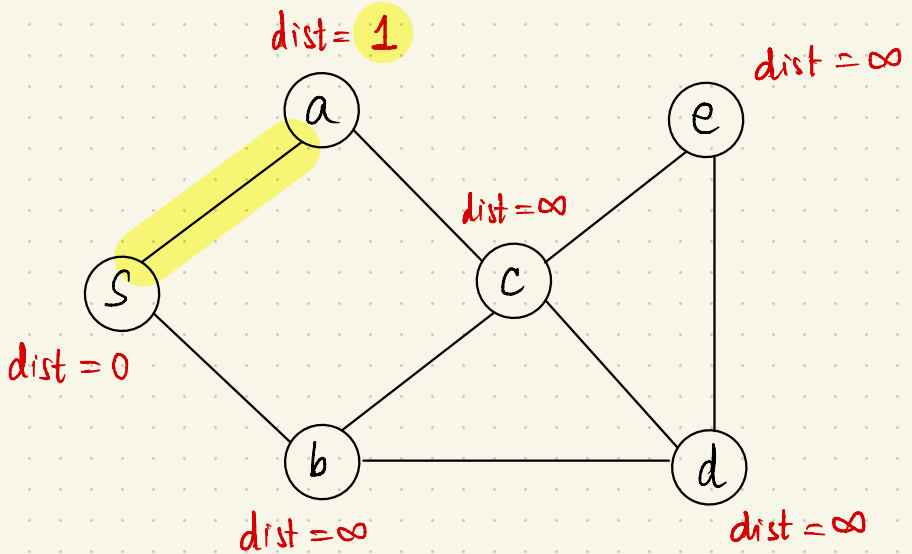
$Q = \cancel{\{\}}$

SHORTEST PATHS



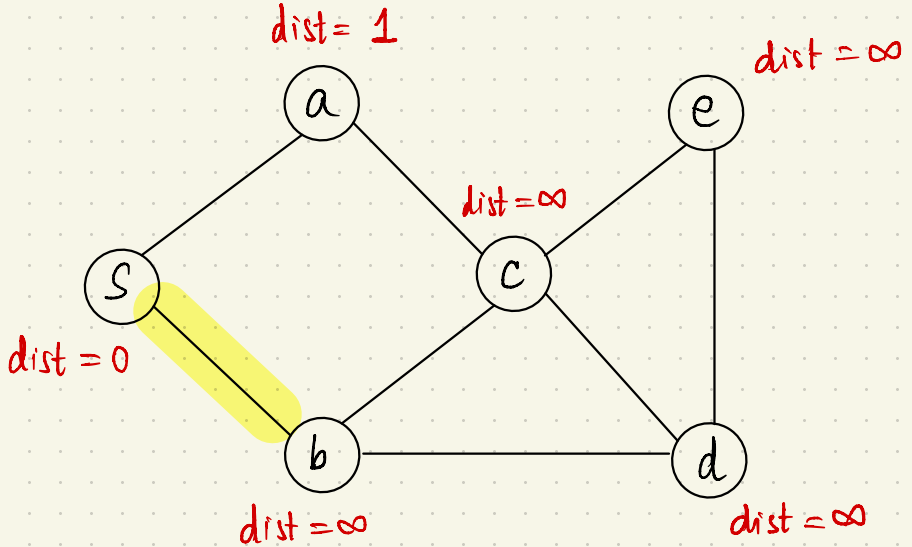
$Q = \cancel{s}\ a$

SHORTEST PATHS



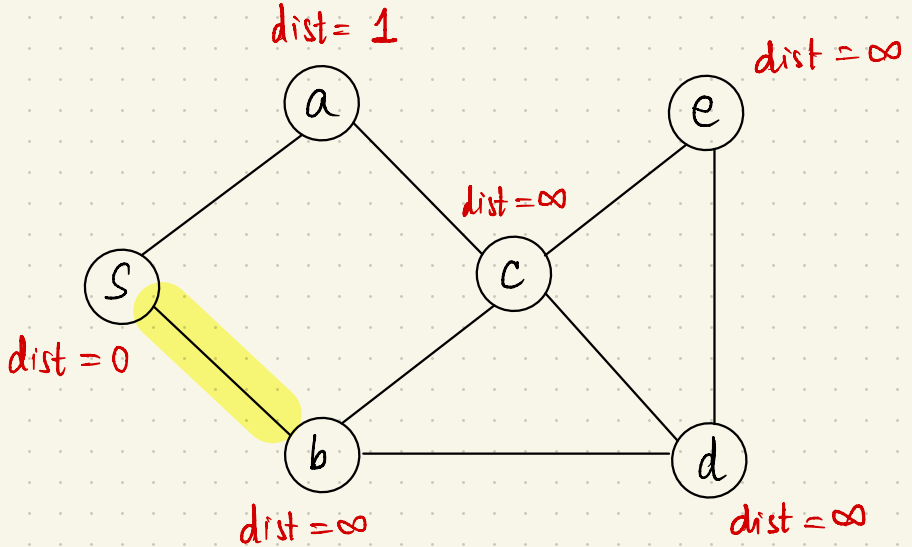
$Q = \cancel{s}\ a$

SHORTEST PATHS



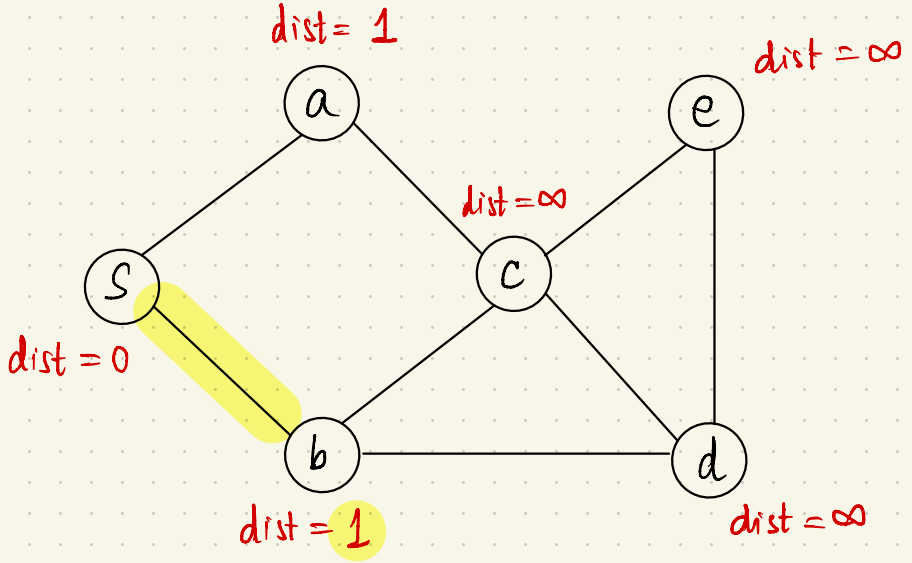
$Q = \cancel{s}\ a$

SHORTEST PATHS



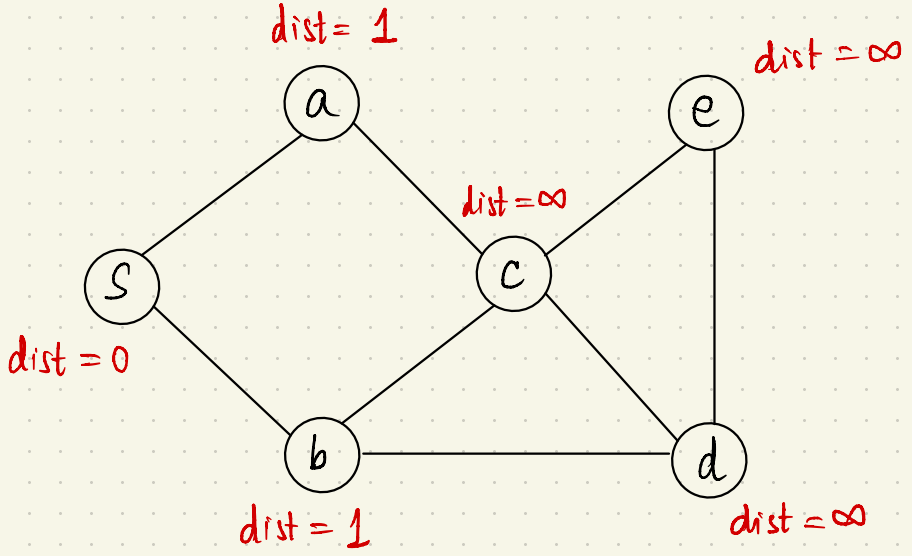
$Q = \cancel{S}\ a\ b$

SHORTEST PATHS



$Q = \cancel{s}\ a\ b$

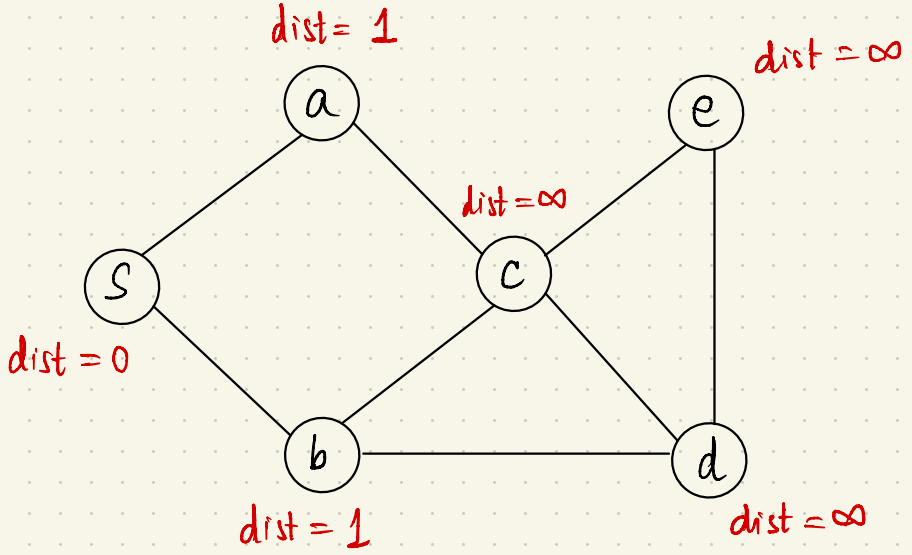
SHORTEST PATHS



$Q = \cancel{s}\ a\ b$

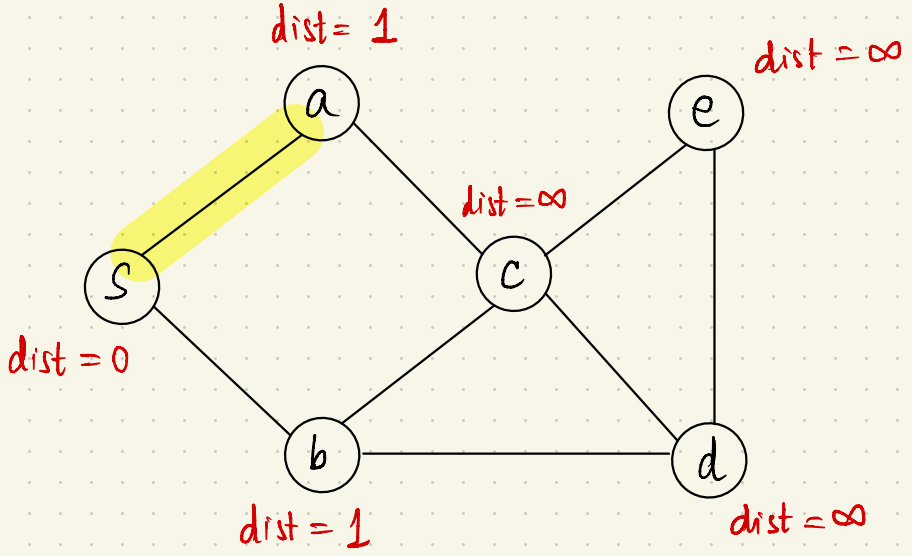
fully explored

SHORTEST PATHS



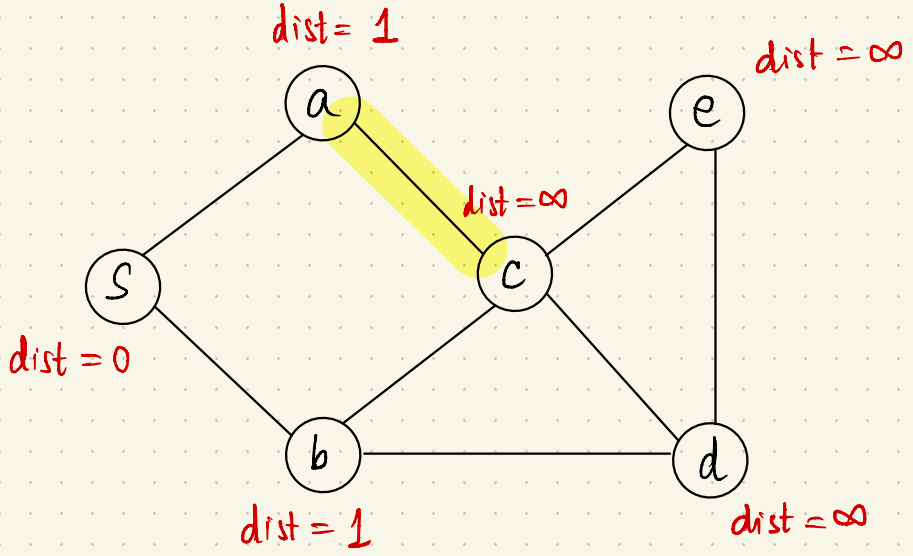
$Q = \cancel{S}\ \cancel{a}\ b$

SHORTEST PATHS



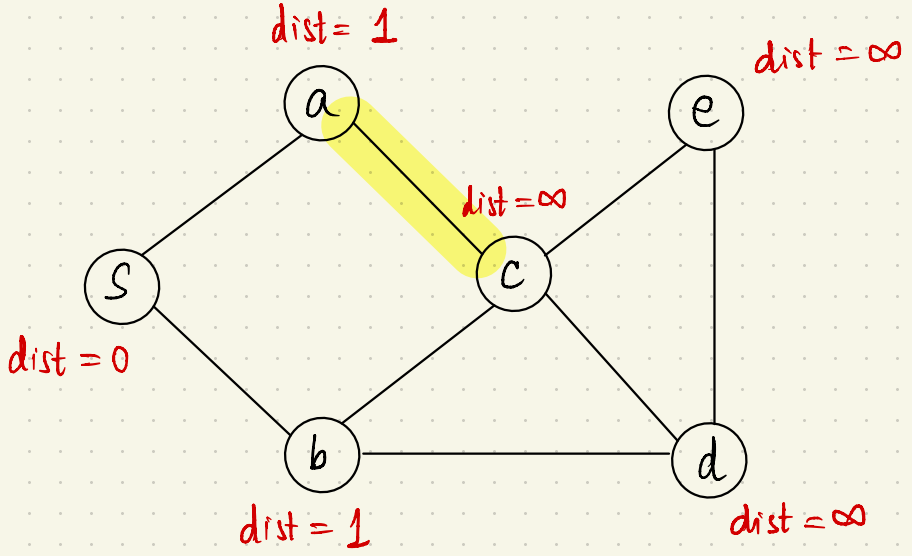
$Q = \cancel{s}\ \cancel{a}\ b$

SHORTEST PATHS



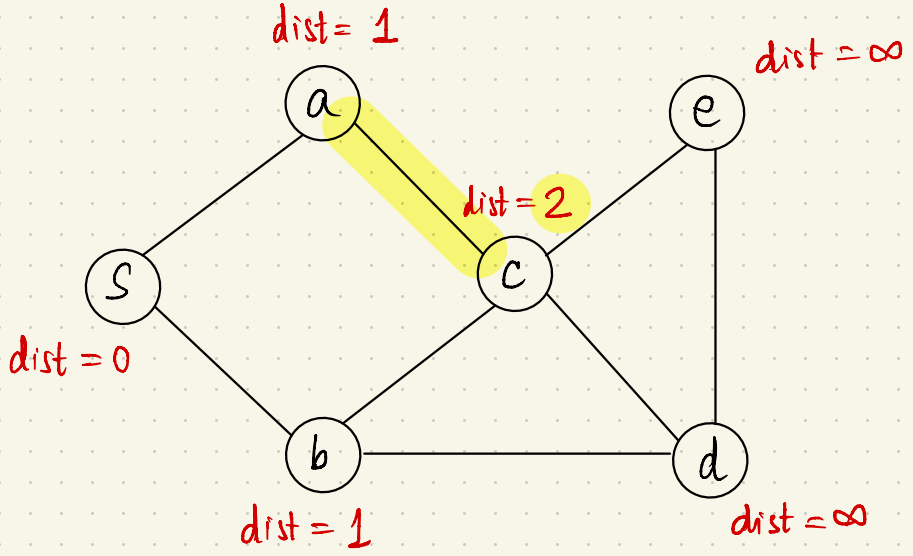
$Q = \cancel{s}\ \cancel{a}\ b$

SHORTEST PATHS



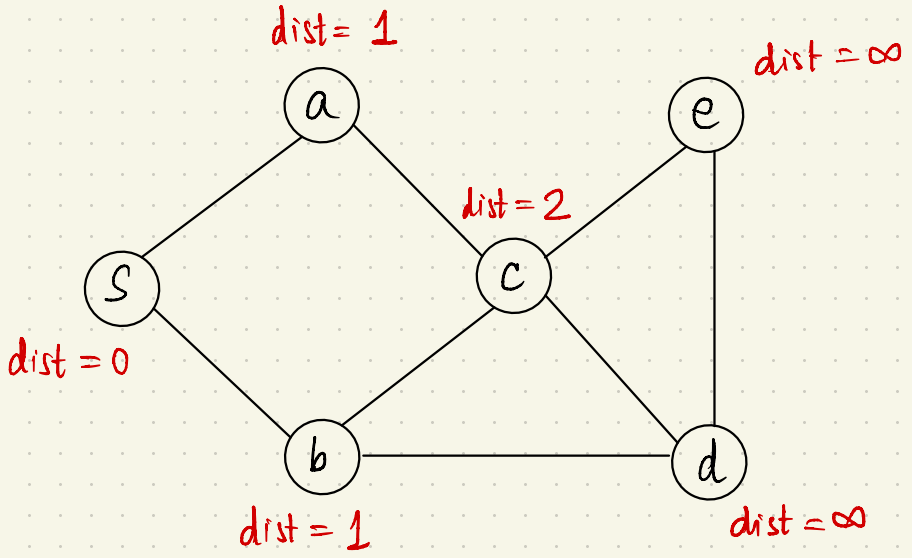
$Q = \not{s}\ \not{a}\ b\ c$

SHORTEST PATHS



$Q = \not{S}\ \not{a}\ b\ c$

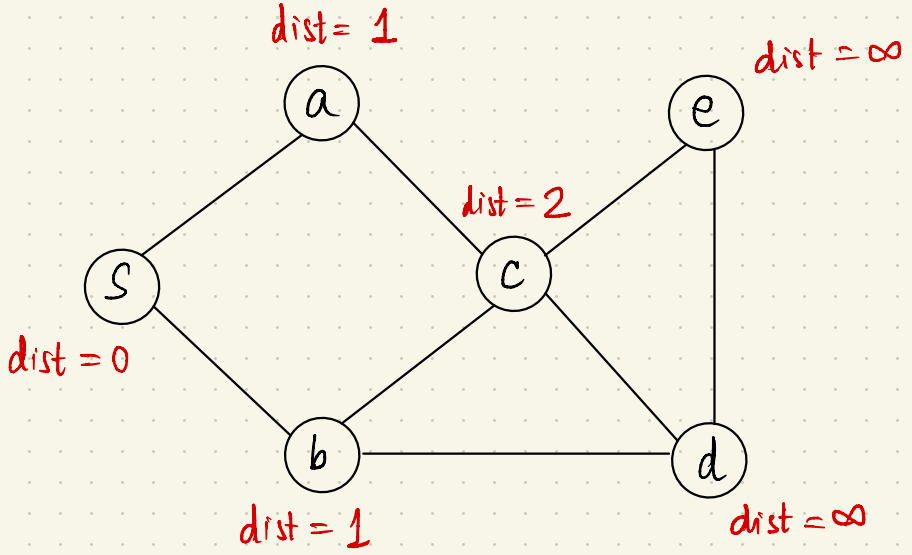
SHORTEST PATHS



$Q = \cancel{S}\ \cancel{a}\ b\ c$

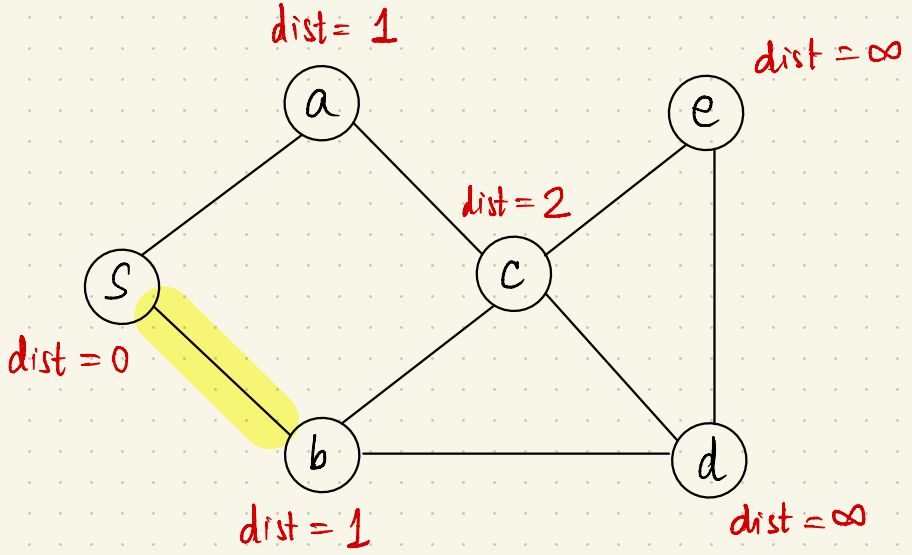
fully explored

SHORTEST PATHS



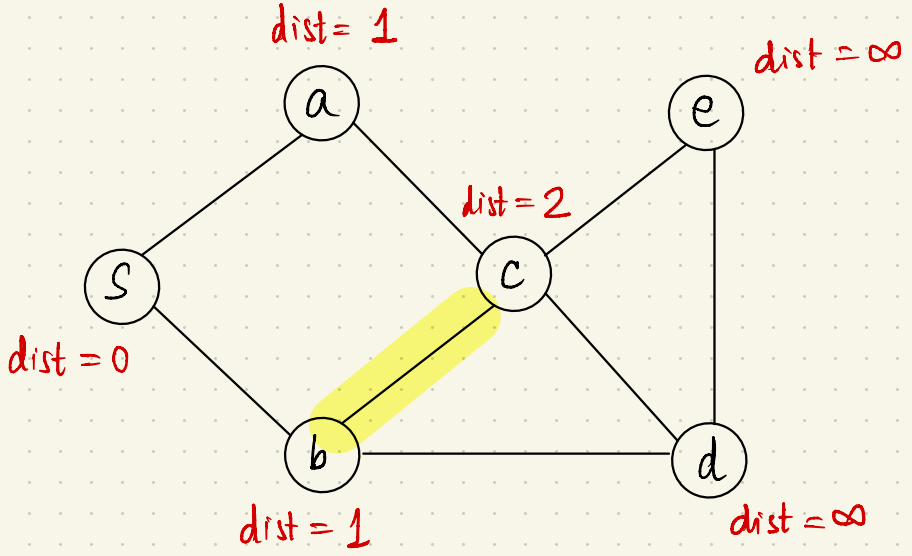
Q = ~~s~~ ~~a~~ ~~b~~ c

SHORTEST PATHS



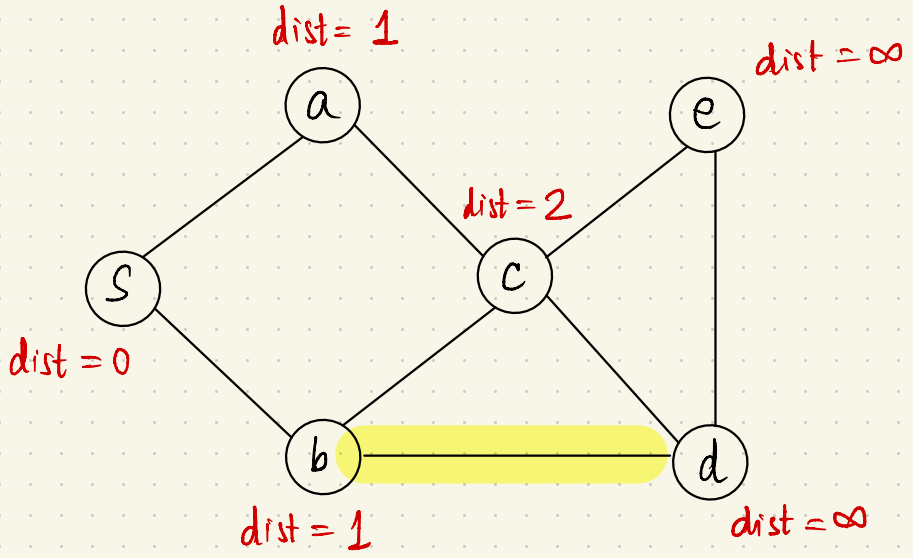
Q = ~~S~~ ~~a~~ ~~b~~ c

SHORTEST PATHS



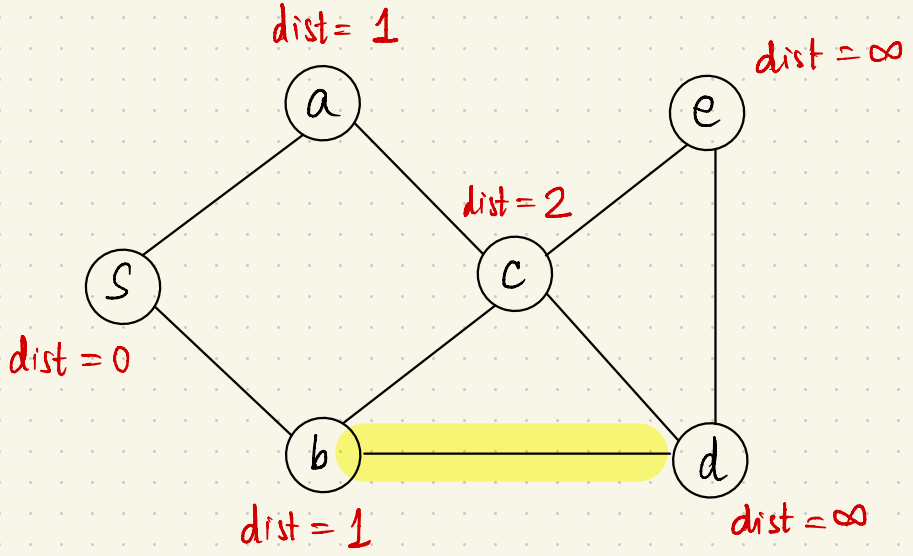
Q = ~~s a b~~ c

SHORTEST PATHS



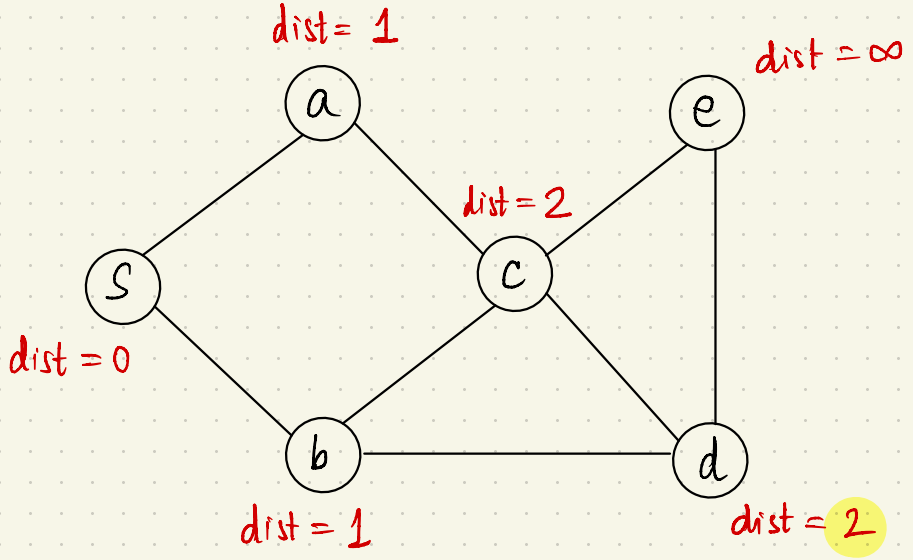
Q = ~~S~~ ~~a~~ ~~b~~ c

SHORTEST PATHS



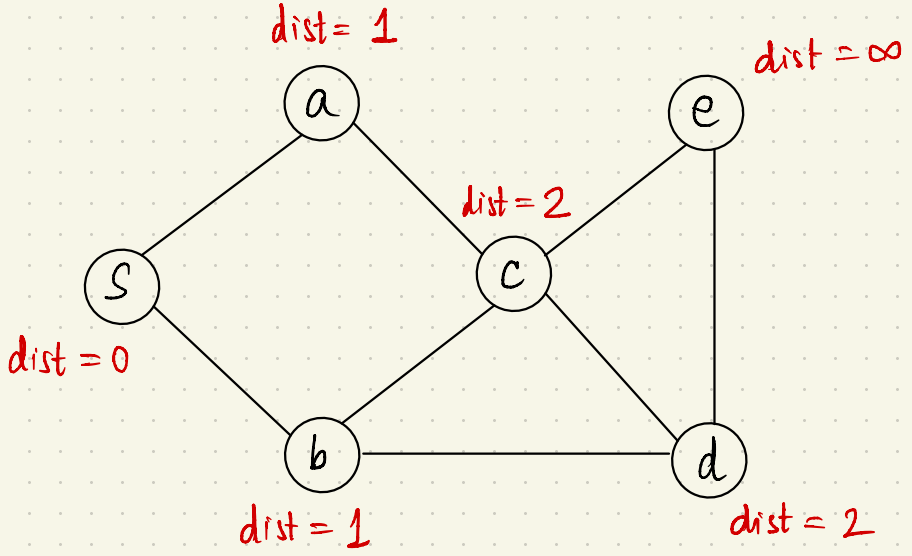
$Q = \not{S}\ \not{a}\ \not{b}\ c\ d$

SHORTEST PATHS



$Q = \not{s}\ \not{a}\ \not{b}\ c\ d$

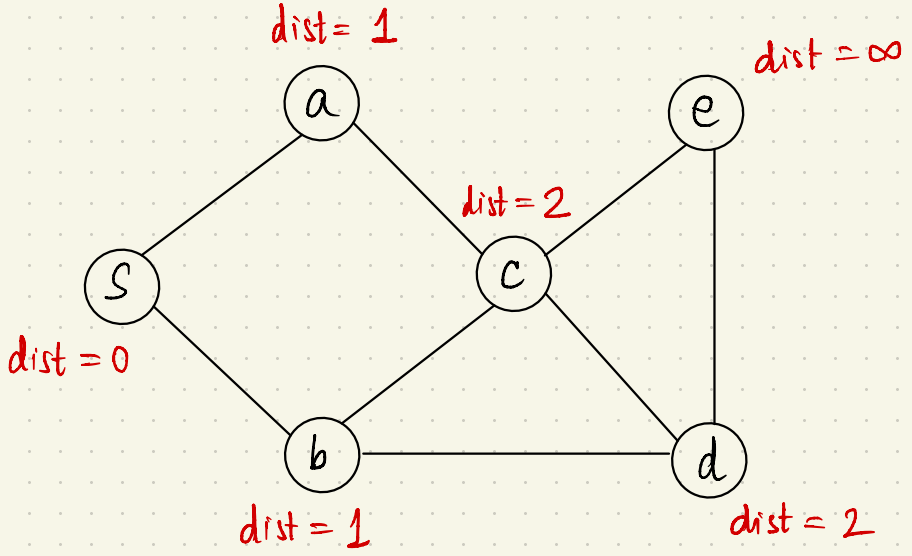
SHORTEST PATHS



$Q = \cancel{S}\ \cancel{a}\ \cancel{b}\ c\ d$

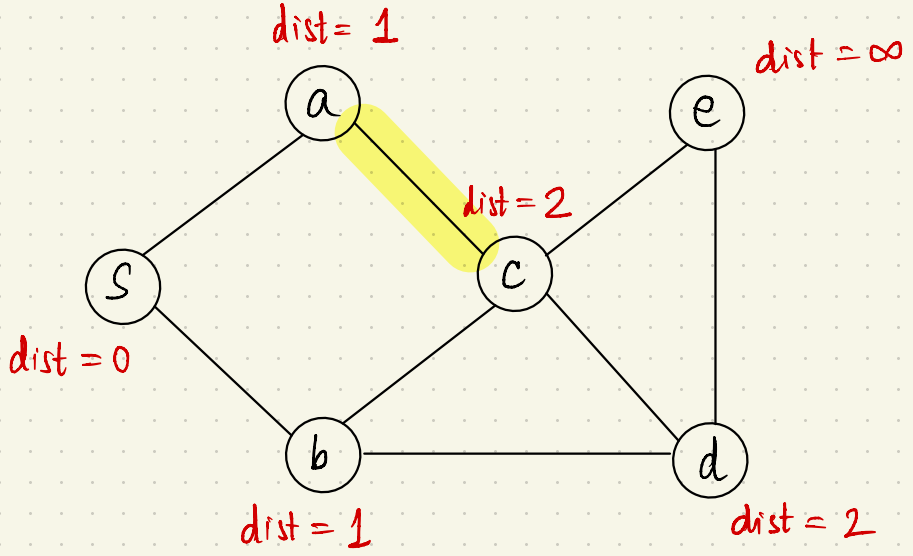
fully explored

SHORTEST PATHS



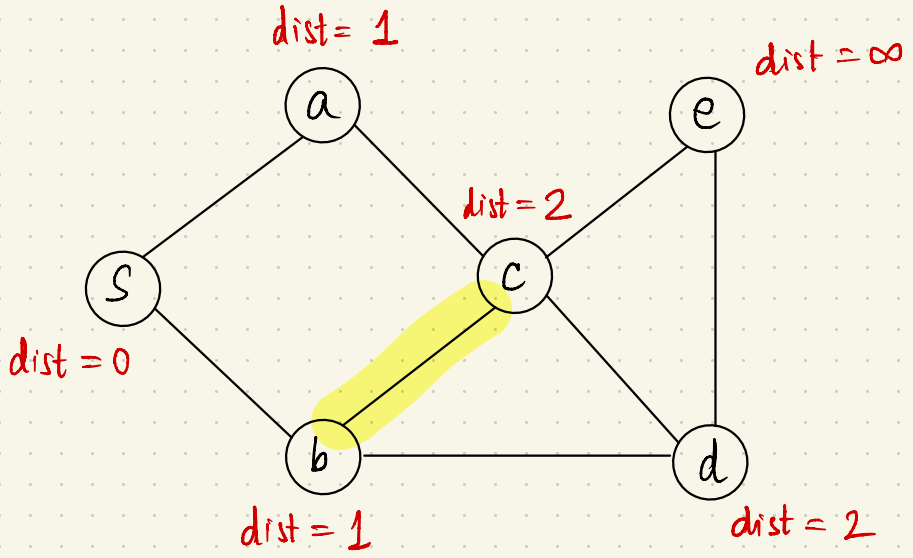
$Q = \not{s}\ \not{a}\ \not{b}\ \not{c}\ d$

SHORTEST PATHS



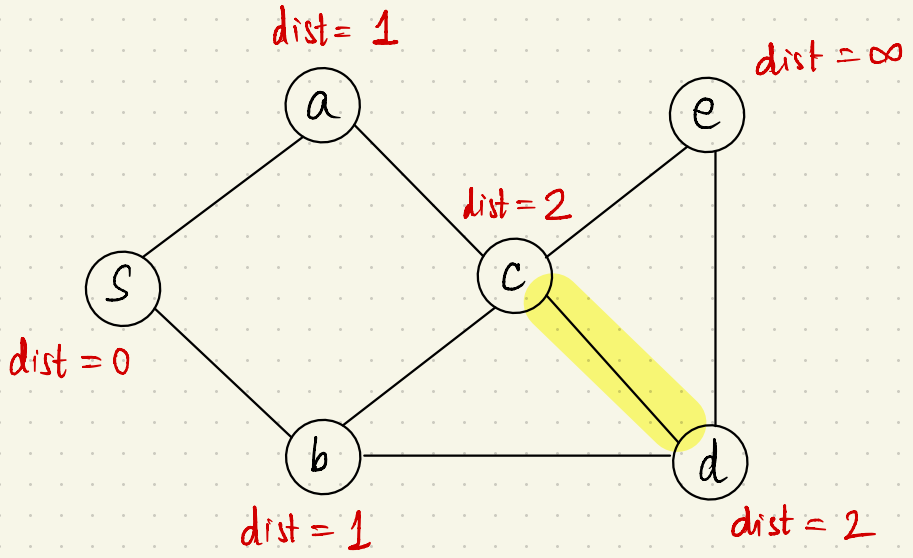
$Q = \not{s}\ \not{a}\ \not{b}\ \not{c}\ d$

SHORTEST PATHS



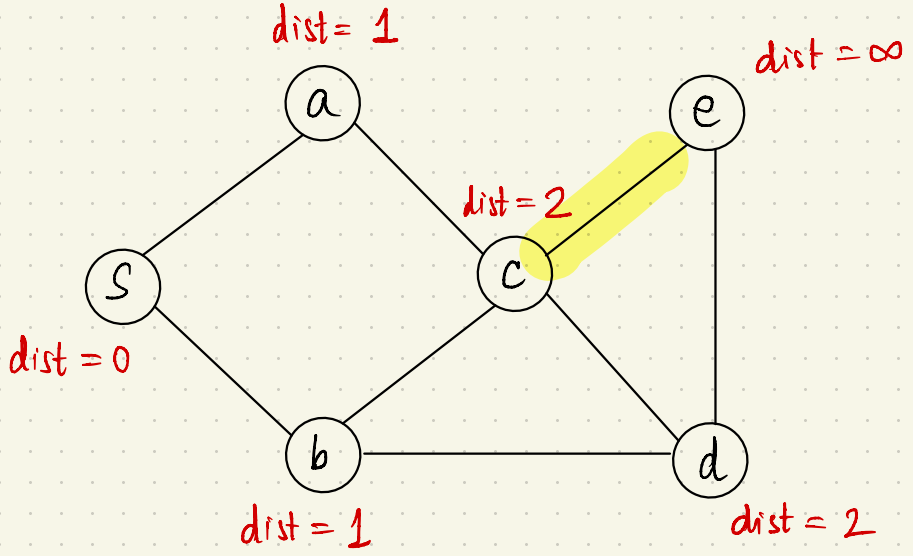
$Q = \not{s}\ \not{a}\ \not{b}\ \not{c}\ d$

SHORTEST PATHS



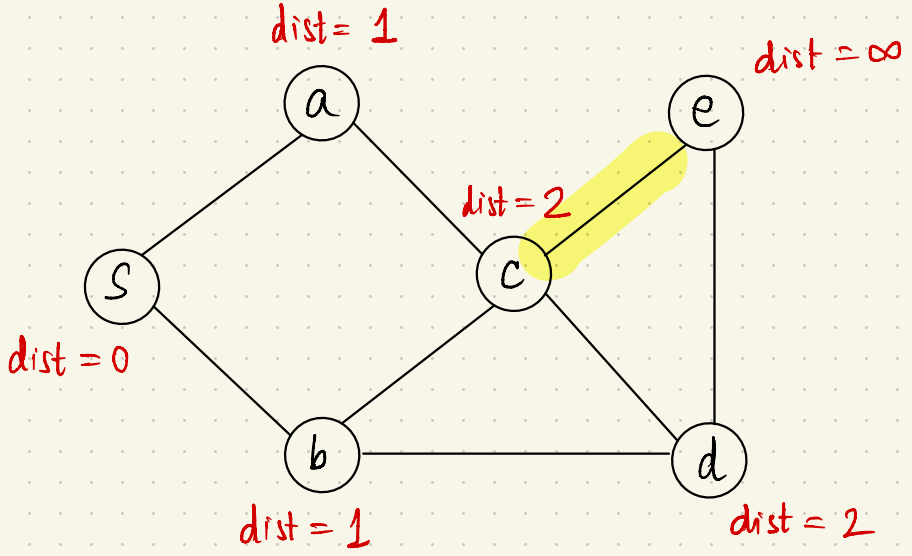
$Q = \cancel{s}\,\cancel{a}\,\cancel{b}\,\cancel{c}\,d$

SHORTEST PATHS



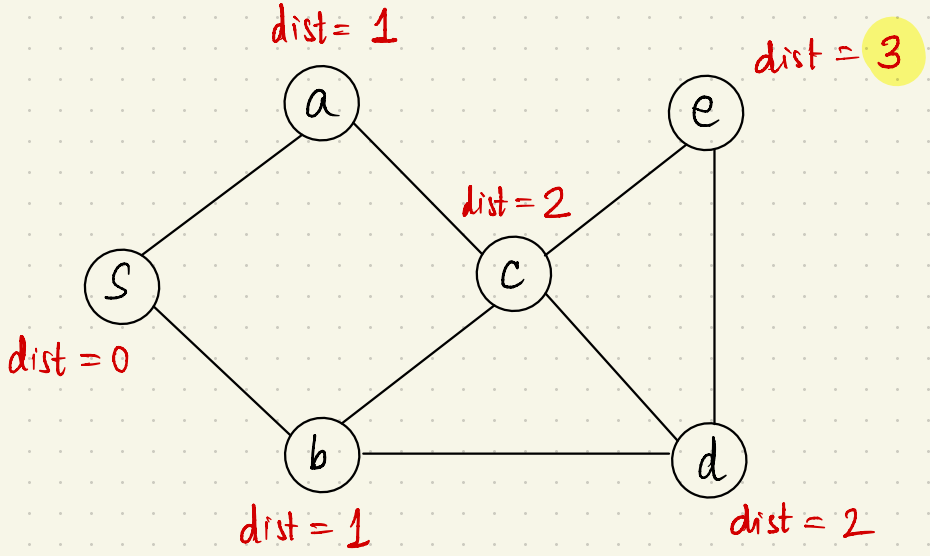
$Q = \cancel{s}\,\cancel{a}\,\cancel{b}\,\cancel{c}\,d$

SHORTEST PATHS



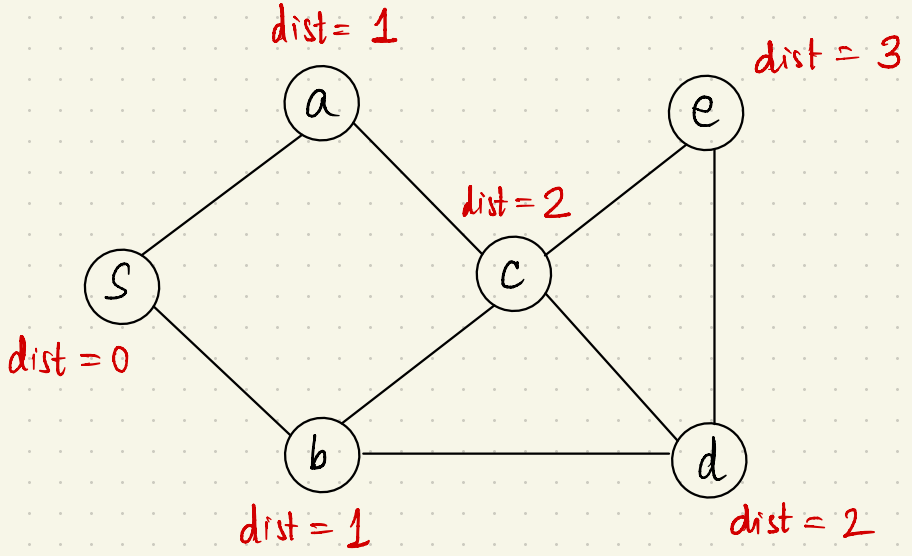
Q = ~~s a b c~~ d e

SHORTEST PATHS



Q = ~~S a b c~~ d e

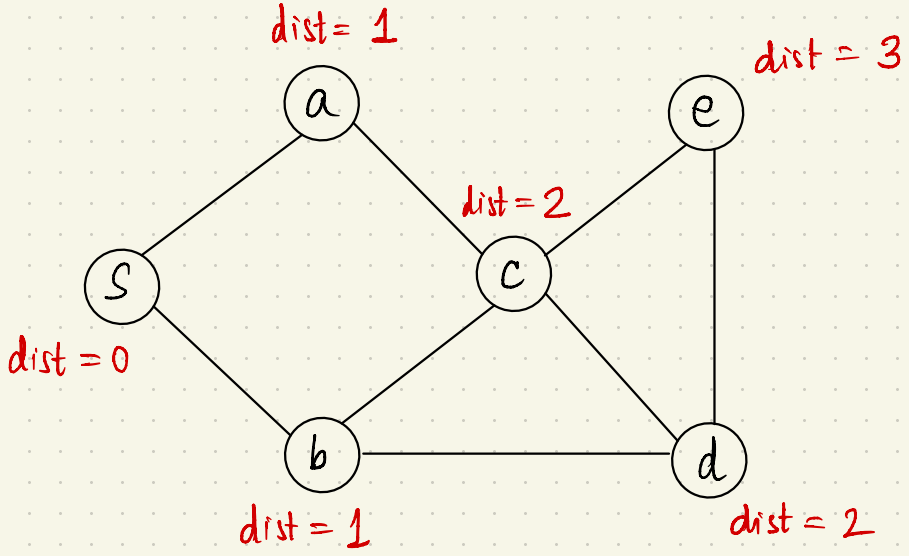
SHORTEST PATHS



Q = ~~S a b c~~ d e

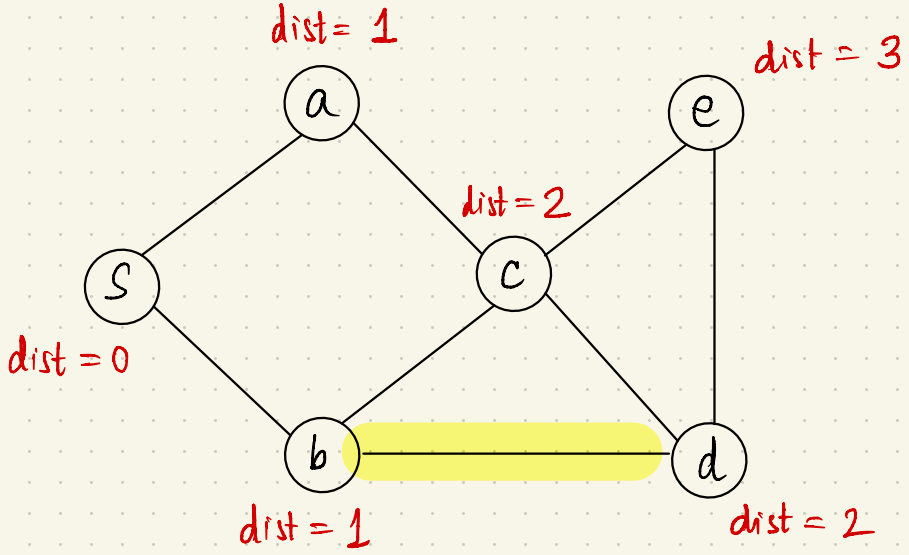
fully explored

SHORTEST PATHS



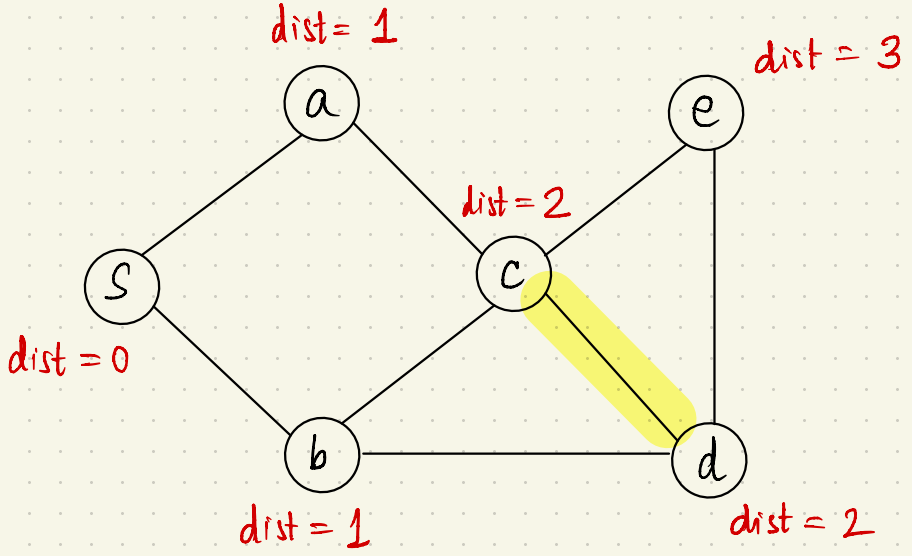
Q = ~~s a b c d~~ e

SHORTEST PATHS



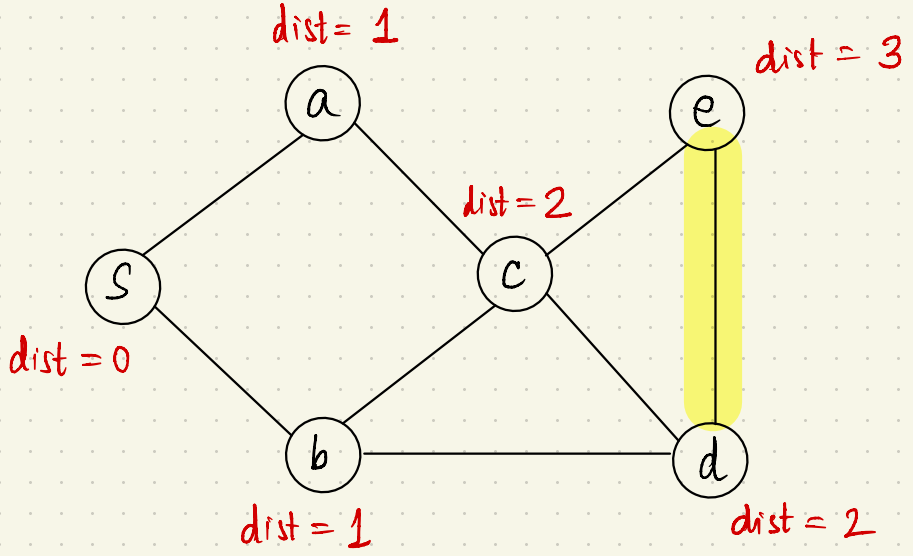
Q = ~~S a b c d~~ e

SHORTEST PATHS



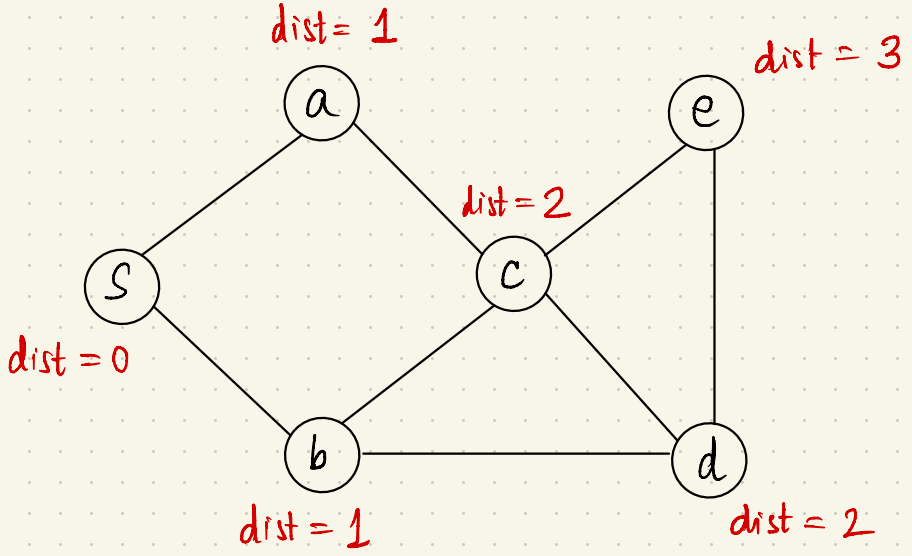
Q = ~~S a b c d~~ e

SHORTEST PATHS



Q = ~~S a b c d~~ e

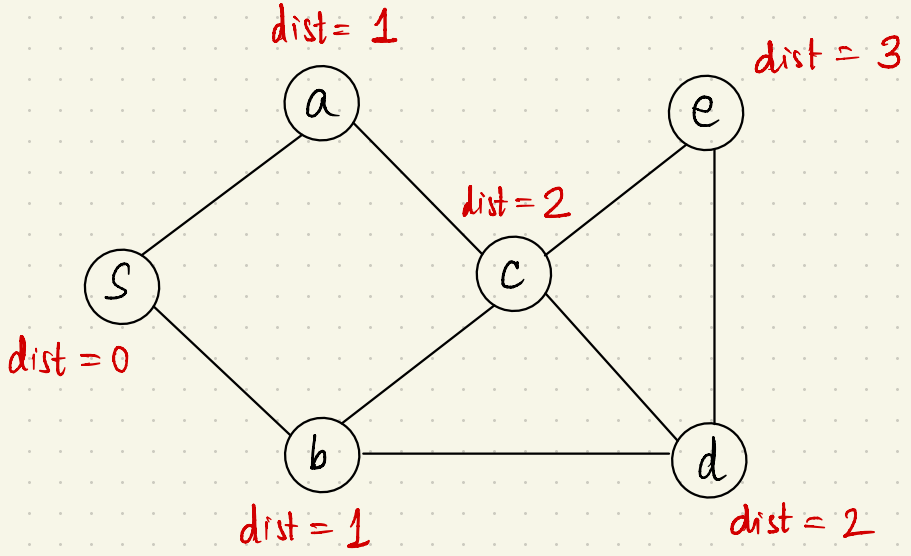
SHORTEST PATHS



Q = ~~S a b c d~~ e

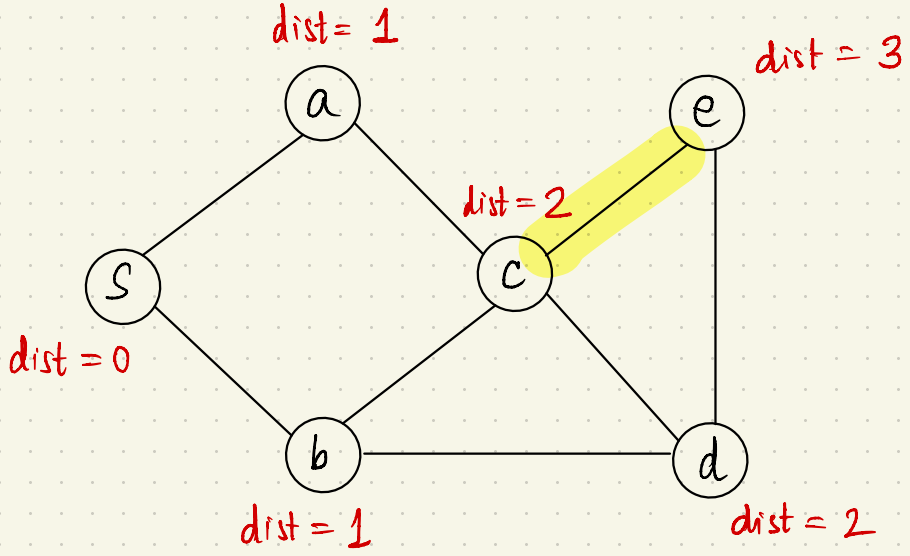
fully explored

SHORTEST PATHS



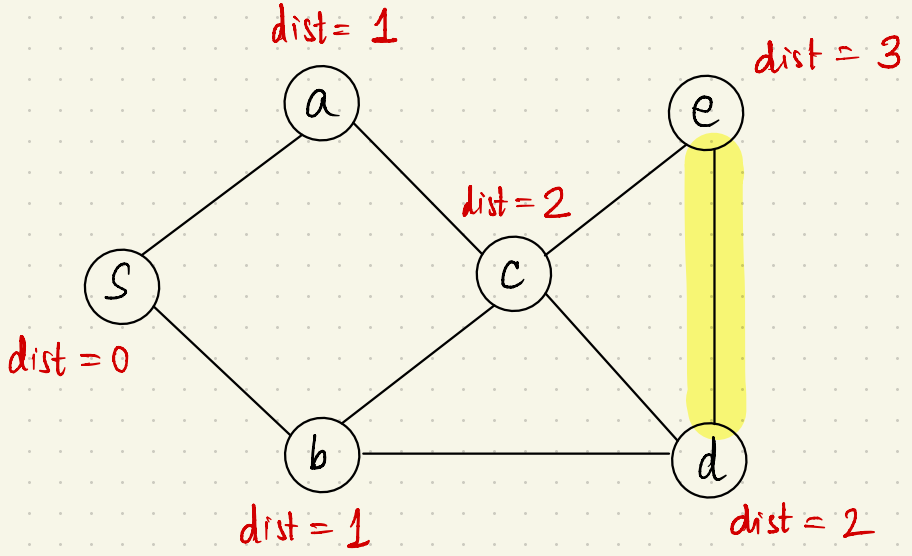
Q = ~~S a b c d e~~

SHORTEST PATHS



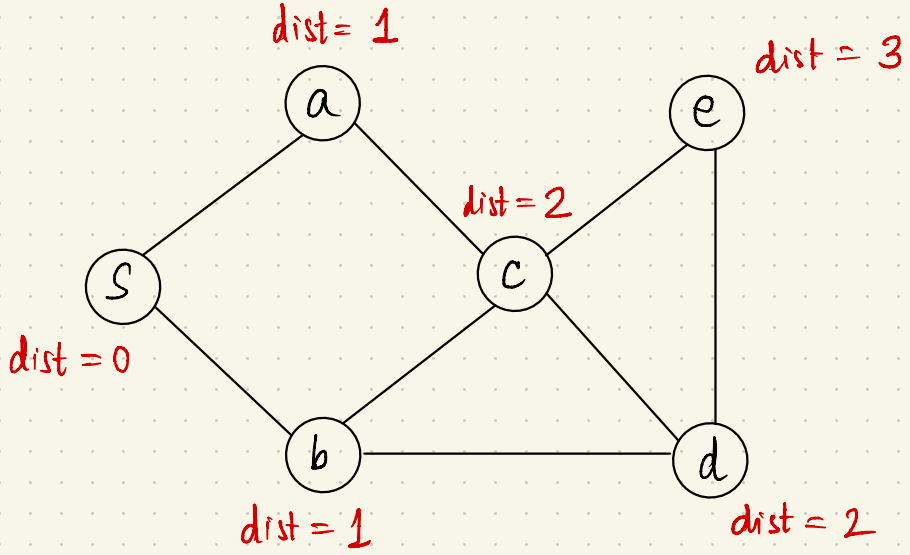
Q = ~~S a b c d e~~

SHORTEST PATHS



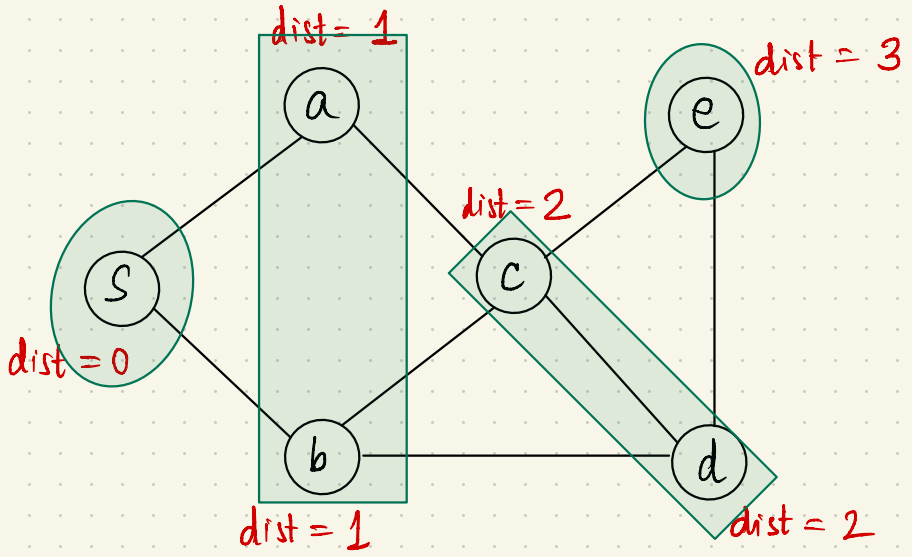
Q = ~~S a b c d e~~

SHORTEST PATHS

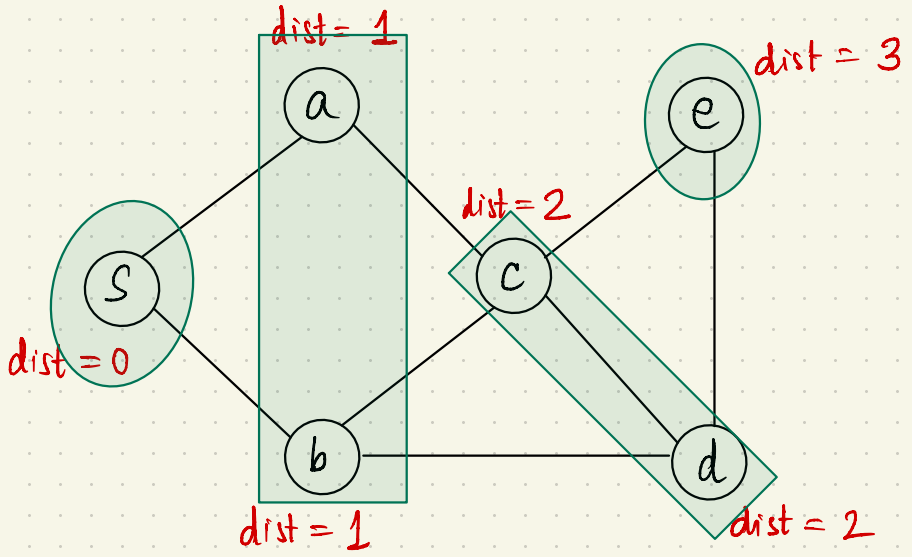


Q = ~~S a b c d e~~ done!

SHORTEST PATHS

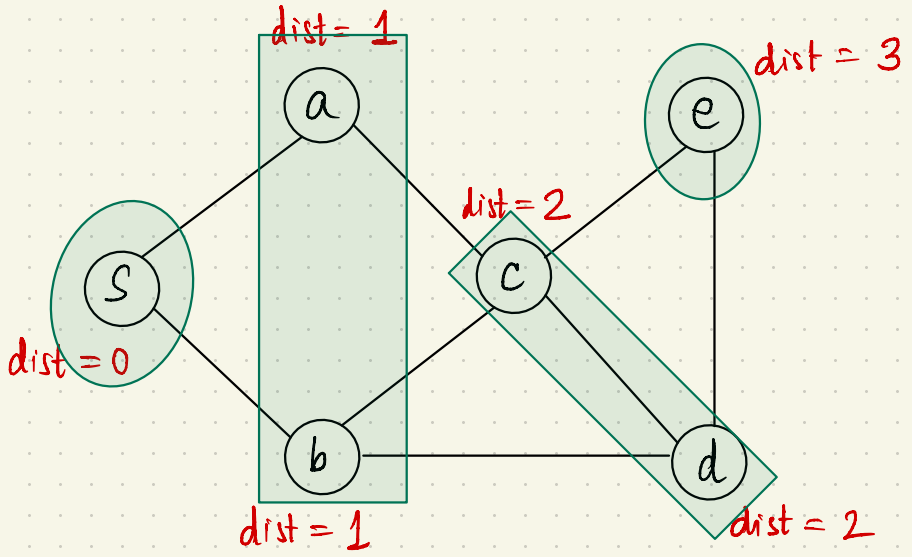


SHORTEST PATHS



Claim: at termination, $dist(v) = i \iff v$ is in i^{th} layer

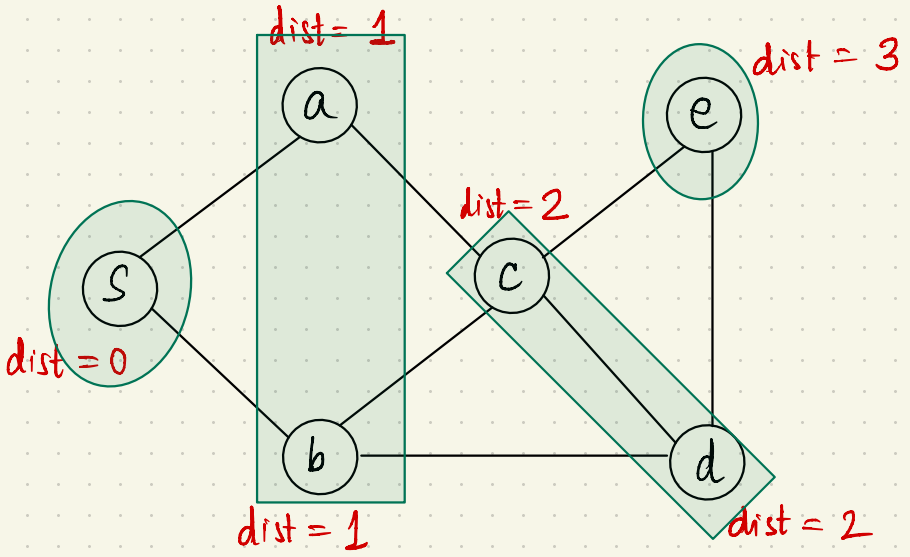
SHORTEST PATHS



Claim: at termination, $dist(v) = i \iff v$ is in i^{th} layer

shortest path $s \to v$ has i edges

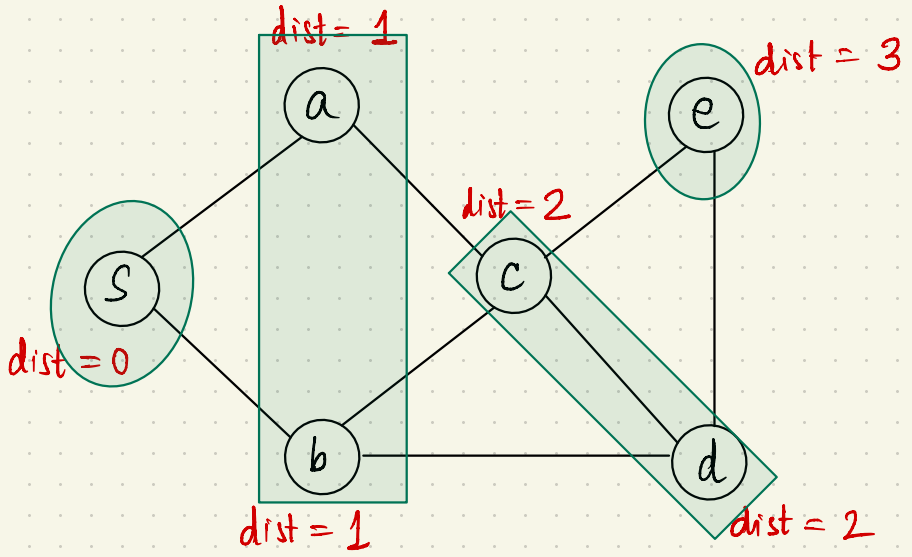
SHORTEST PATHS



Claim: at termination, $dist(v) = i \iff v$ is in i^{th} layer

Proof: Exercise (by induction)

SHORTEST PATHS



Claim: The augmented BFS has running time $O(m+n)$.

APPLICATIONS OF BFS

Shortest paths

Connected Components

Applications of BFS

Shortest paths

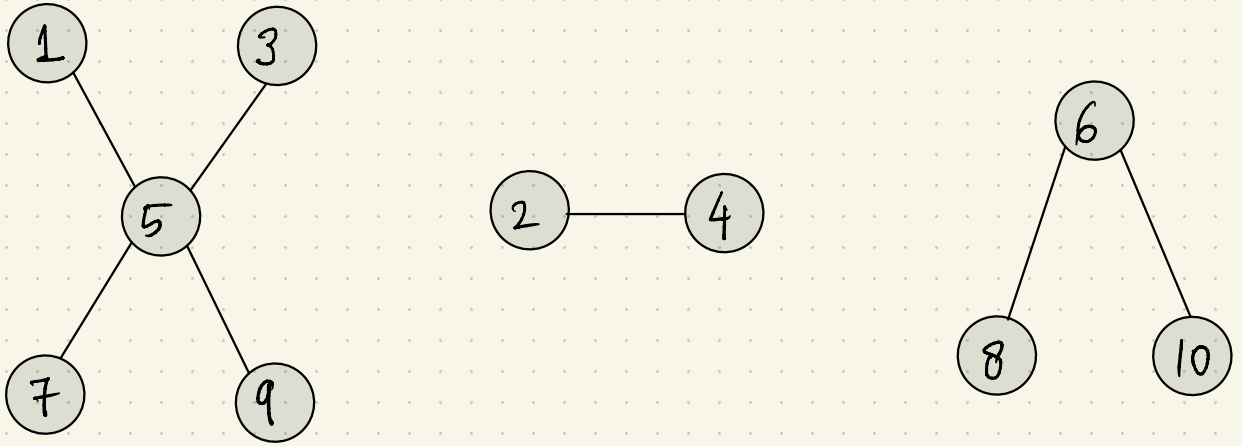
Connected Components

CONNECTED COMPONENTS

Undirected graph $G = (V, E)$

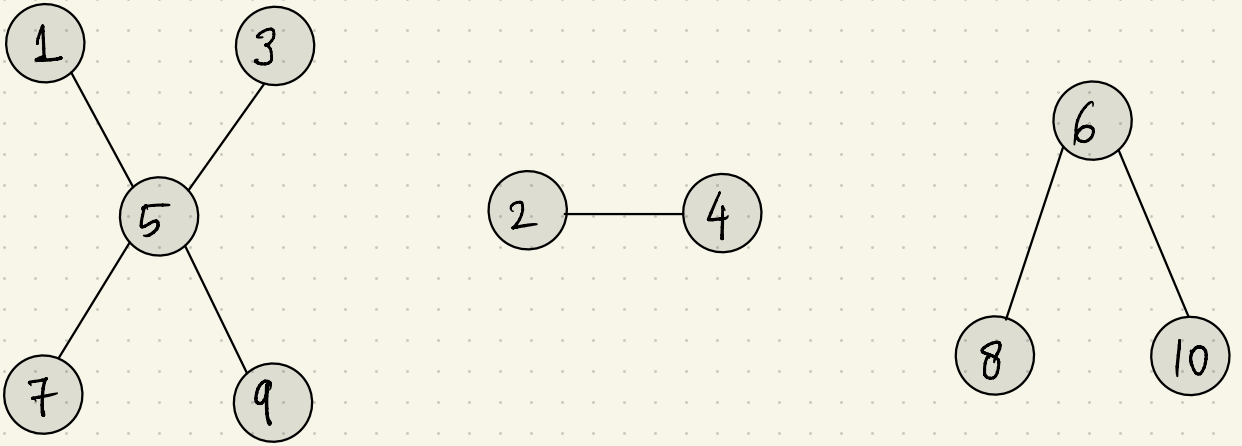
CONNECTED COMPONENTS

Undirected graph $G = (V, E)$



CONNECTED COMPONENTS

Undirected graph $G = (V, E)$

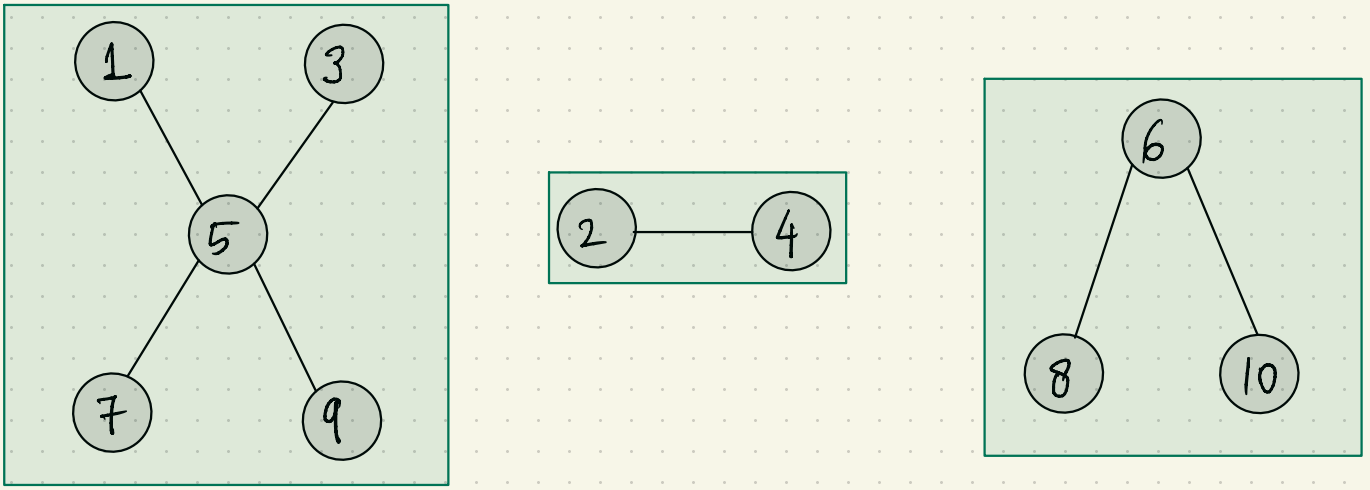


Connected components : maximal subsets of reachable vertices

(or equivalence class)

CONNECTED COMPONENTS

Undirected graph $G = (V, E)$



Connected components : maximal subsets of reachable vertices

(or equivalence class)

CONNECTED COMPONENTS

input: an undirected graph $G = (V, E)$

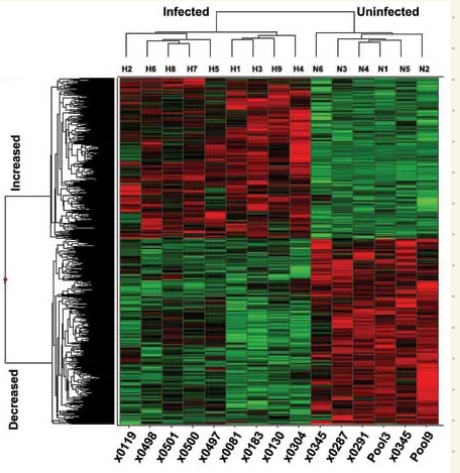
output: identify the connected components of G

CONNECTED COMPONENTS

input: an undirected graph $G = (V, E)$

output: identify the connected components of G

Detecting network failures



Genome clustering

Connected components via BFS

CONNECTED COMPONENTS via BFS

mark all vertices $1, 2, \ldots, n$ as unexplored

initialize CC count := 0

CONNECTED COMPONENTS via BFS

mark all vertices $1, 2, \ldots, n$ as unexplored

initialize CC count := 0

for $i = 1$ to n // try all vertices

CONNECTED COMPONENTS via BFS

```
mark all vertices 1, 2, ..., n as unexplored
initialize CC count := 0
for i = 1 to n                    // try all vertices
    if i is unexplored
        increment CC count by 1
```

CONNECTED COMPONENTS via BFS

mark all vertices $1, 2, \ldots, n$ as unexplored

initialize CC count := 0

for i = 1 to n // try all vertices

  if i is unexplored

    increment CC count by 1

    BFS (G, i)

    (while updating CC number of a vertex extracted from queue

CONNECTED COMPONENTS via BFS

mark all vertices 1, 2, ..., n as unexplored

initialize CC count := 0

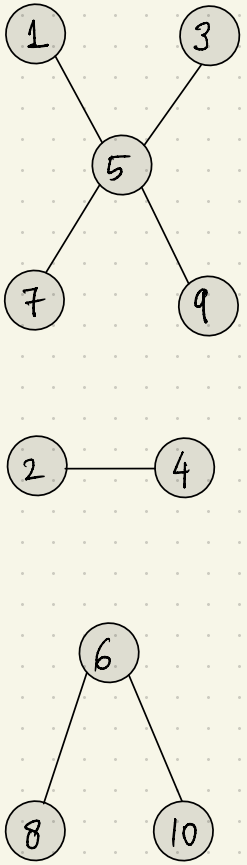
for i = 1 to n // try all vertices

    if i is unexplored

        increment CC count by 1

        BFS (G, i)

        (while updating CC number of a vertex extracted from queue



CONNECTED COMPONENTS via BFS

mark all vertices $1, 2, \ldots, n$ as unexplored

initialize CC count := 0

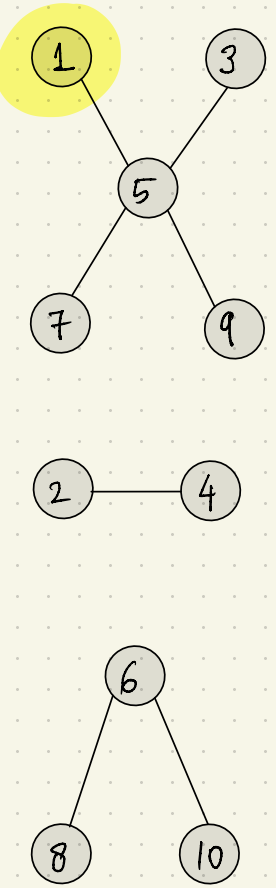
for i = 1 to n // try all vertices

  if i is unexplored

    increment CC count by 1

    BFS (G, i)

    (while updating CC number of a vertex extracted from queue



CONNECTED COMPONENTS via BFS

mark all vertices $1, 2, \ldots, n$ as unexplored

initialize CC count := 0

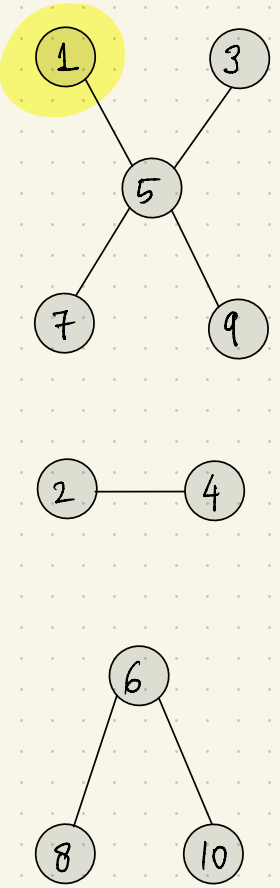
for i = 1 to n // try all vertices

  if i is unexplored

    increment CC count by 1

    BFS (G, i)

    (while updating CC number of a vertex extracted from queue



CONNECTED COMPONENTS via BFS

mark all vertices 1, 2, ..., n as unexplored

initialize CC count := 0

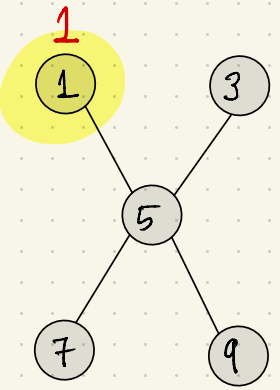
for i = 1 to n // try all vertices

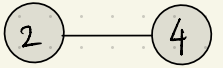
  if i is unexplored

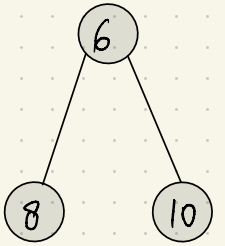
    increment CC count by 1

    BFS (G, i)

    (while updating CC number of a vertex extracted from queue







CONNECTED COMPONENTS via BFS

mark all vertices 1, 2, ..., n as unexplored

initialize CC count := 0

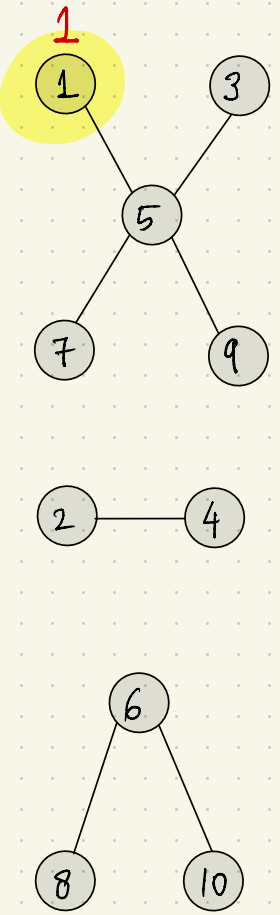
for i = 1 to n // try all vertices

  if i is unexplored

    increment CC count by 1

    BFS (G, i)

      while updating CC number of a vertex extracted from queue



CONNECTED COMPONENTS via BFS

mark all vertices 1, 2, ..., n as unexplored

initialize CC count := 0

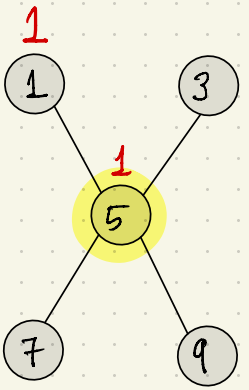
for i = 1 to n // try all vertices

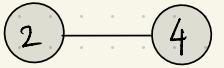
  if i is unexplored

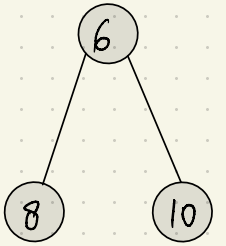
    increment CC count by 1

    BFS (G, i)

      ↳ while updating CC number of a vertex extracted from queue







CONNECTED COMPONENTS via BFS

mark all vertices 1, 2, ..., n as unexplored

initialize CC count := 0

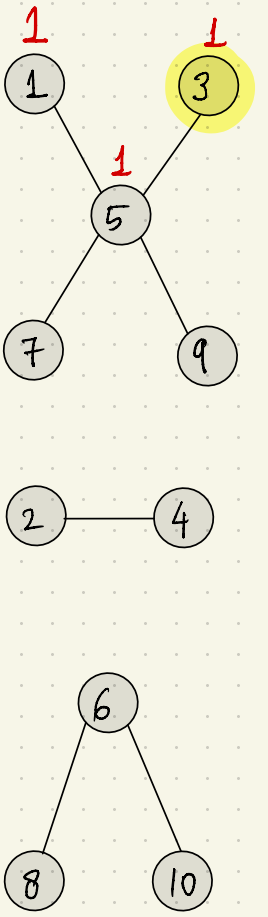
for i = 1 to n // try all vertices

    if i is unexplored

        increment CC count by 1

        BFS (G, i)

            (while updating CC number of a vertex extracted from queue



CONNECTED COMPONENTS via BFS

mark all vertices 1, 2, ..., n as unexplored

initialize CC count := 0

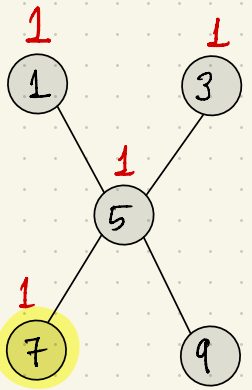
for i = 1 to n // try all vertices

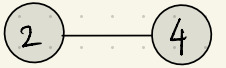
    if i is unexplored

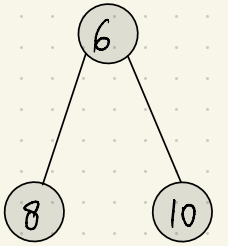
        increment CC count by 1

        BFS (G, i)

            while updating CC number of a vertex extracted from queue







CONNECTED COMPONENTS via BFS

mark all vertices 1, 2, ..., n as unexplored

initialize CC count := 0

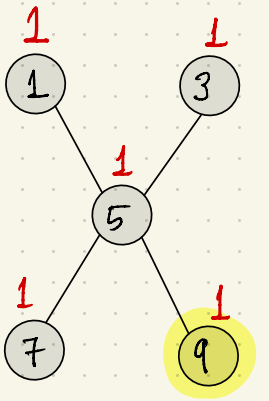
for i = 1 to n // try all vertices

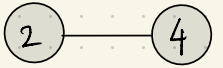
    if i is unexplored

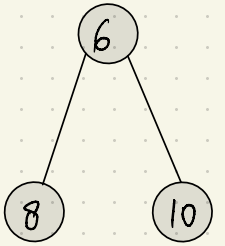
        increment CC count by 1

        BFS (G, i)

            while updating CC number of a vertex extracted from queue







CONNECTED COMPONENTS via BFS

mark all vertices 1, 2, ..., n as unexplored

initialize CC count := 0

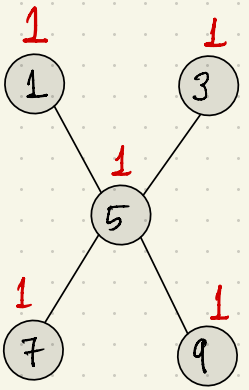
for i = 1 to n // try all vertices

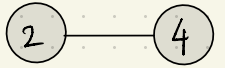
  if i is unexplored

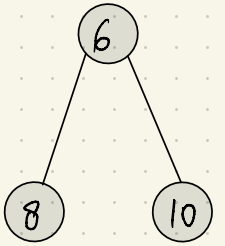
    increment CC count by 1

    BFS (G, i)

      while updating CC number of a vertex extracted from queue







CONNECTED COMPONENTS via BFS

mark all vertices 1, 2, ..., n as unexplored

initialize CC count := 0

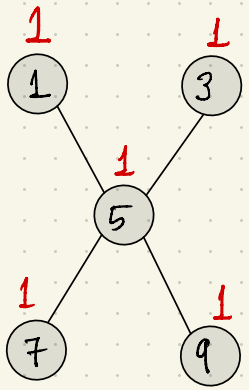
for i = 1 to n // try all vertices

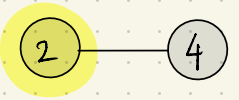
    if i is unexplored

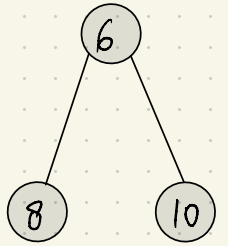
        increment CC count by 1

        BFS (G, i)

        (while updating CC number of a vertex extracted from queue







CONNECTED COMPONENTS via BFS

mark all vertices 1, 2, ..., n as unexplored

initialize CC count := 0

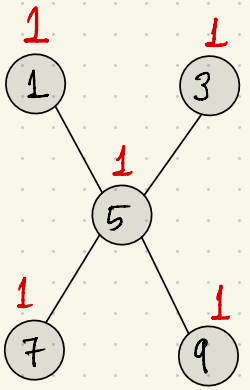
for i = 1 to n // try all vertices

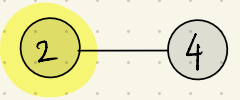
if i is unexplored

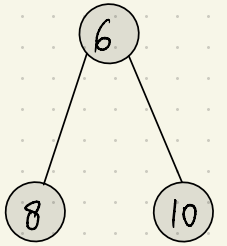
increment CC count by 1

BFS (G, i)

(while updating CC number of a vertex extracted from queue







CONNECTED COMPONENTS via BFS

mark all vertices 1, 2, ..., n as unexplored

initialize CC count := 0

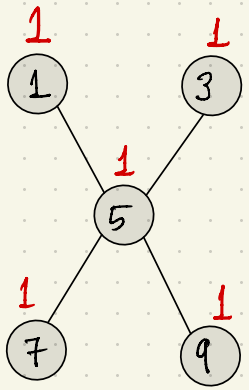
for i = 1 to n // try all vertices

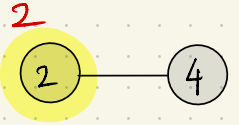
  if i is unexplored

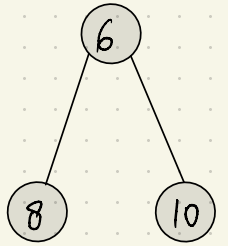
    increment CC count by 1

    BFS (G, i)

    (while updating CC number of a vertex extracted from queue







CONNECTED COMPONENTS via BFS

mark all vertices 1, 2, ..., n as unexplored

initialize CC count := 0

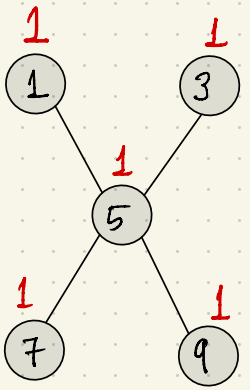
for i = 1 to n // try all vertices

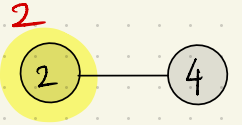
  if i is unexplored

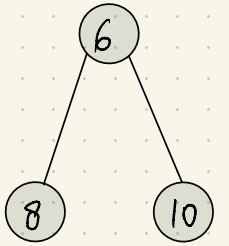
    increment CC count by 1

    BFS (G, i)

      while updating CC number of a vertex extracted from queue







CONNECTED COMPONENTS via BFS

mark all vertices 1, 2, ..., n as unexplored

initialize CC count := 0

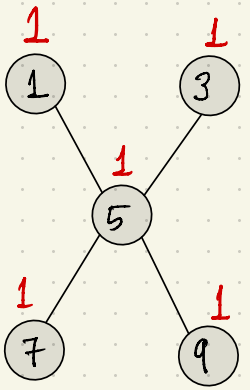
for i = 1 to n // try all vertices

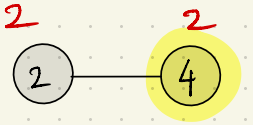
  if i is unexplored

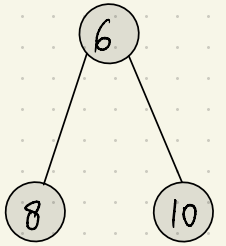
    increment CC count by 1

    BFS (G, i)

      while updating CC number of a vertex extracted from queue







CONNECTED COMPONENTS via BFS

mark all vertices 1, 2, ..., n as unexplored

initialize CC count := 0

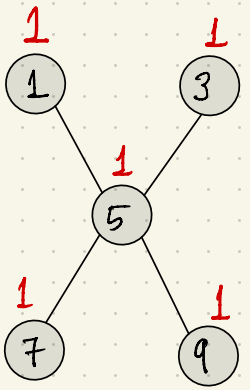
for i = 1 to n // try all vertices

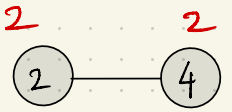
  if i is unexplored

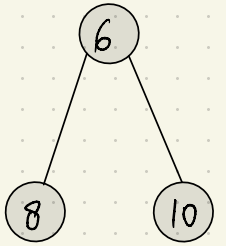
    increment CC count by 1

    BFS (G, i)

      while updating CC number of a vertex extracted from queue







CONNECTED COMPONENTS via BFS

mark all vertices 1, 2, ..., n as unexplored

initialize CC count := 0

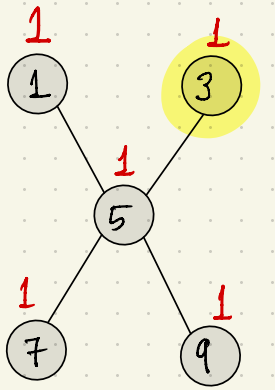
for i = 1 to n // try all vertices

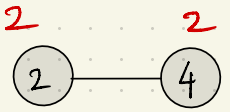
  if i is unexplored

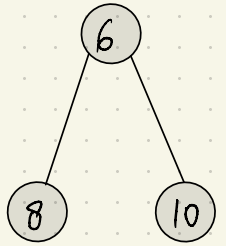
    increment CC count by 1

    BFS (G, i)

    (while updating CC number of a vertex extracted from queue







CONNECTED COMPONENTS via BFS

mark all vertices 1, 2, ..., n as unexplored

initialize CC count := 0

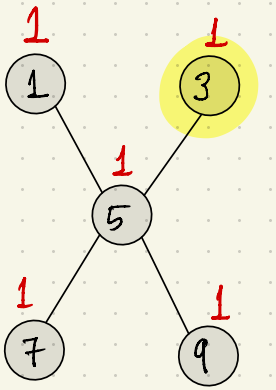
for i = 1 to n // try all vertices

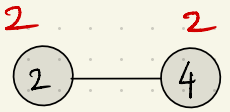
  if i is unexplored

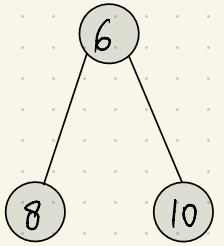
    increment CC count by 1

    BFS (G, i)

    (while updating CC number of a vertex extracted from queue







CONNECTED COMPONENTS via BFS

mark all vertices 1, 2, ..., n as unexplored

initialize CC count := 0

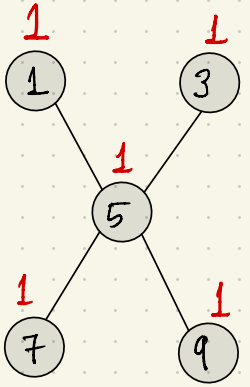
for i = 1 to n // try all vertices

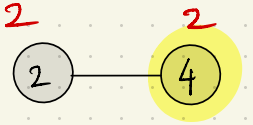
if i is unexplored

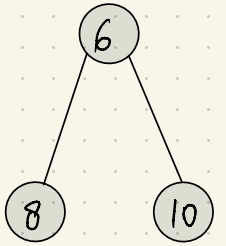
increment CC count by 1

BFS (G, i)

(while updating CC number of a vertex extracted from queue







CONNECTED COMPONENTS via BFS

mark all vertices $1, 2, \ldots, n$ as unexplored

initialize CC count := 0

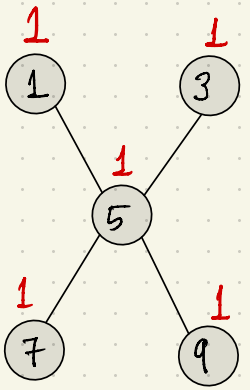
for i = 1 to n // try all vertices

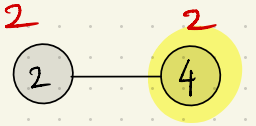
  if i is unexplored

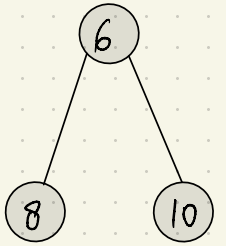
    increment CC count by 1

    BFS (G, i)

    (while updating CC number of a vertex extracted from queue







CONNECTED COMPONENTS via BFS

mark all vertices $1, 2, \ldots, n$ as unexplored

initialize CC count := 0

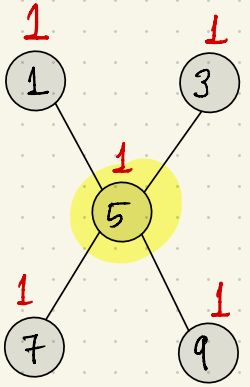
for i = 1 to n // try all vertices

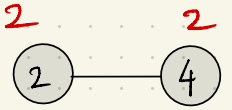
  if i is unexplored

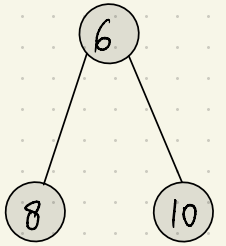
    increment CC count by 1

    BFS (G, i)

    (while updating CC number of a vertex extracted from queue







CONNECTED COMPONENTS via BFS

mark all vertices 1, 2, ..., n as unexplored

initialize CC count := 0

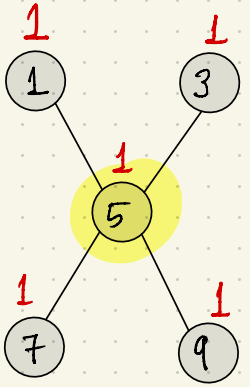
for i = 1 to n // try all vertices

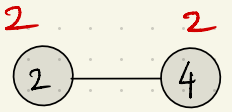
  if i is unexplored

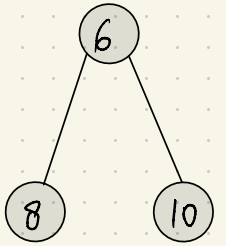
    increment CC count by 1

    BFS (G, i)

    (while updating CC number of a vertex extracted from queue







CONNECTED COMPONENTS via BFS

mark all vertices 1, 2, ..., n as unexplored

initialize CC count := 0

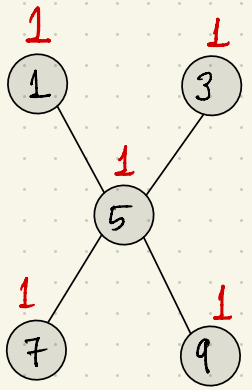
for i = 1 to n // try all vertices

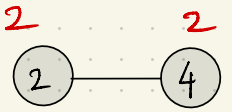
if i is unexplored

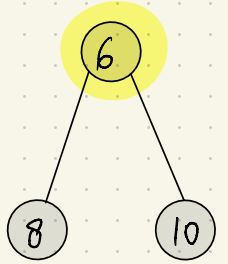
increment CC count by 1

BFS (G, i)

(while updating CC number of a vertex extracted from queue







CONNECTED COMPONENTS via BFS

mark all vertices 1, 2, ..., n as unexplored

initialize CC count := 0

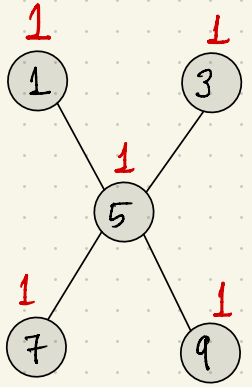
for i = 1 to n // try all vertices

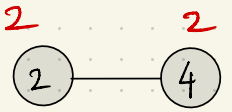
  if i is unexplored

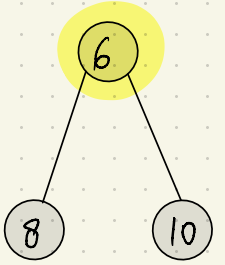
    increment CC count by 1

    BFS (G, i)

    (while updating CC number of a vertex extracted from queue







CONNECTED COMPONENTS via BFS

mark all vertices 1, 2, ..., n as unexplored

initialize CC count := 0

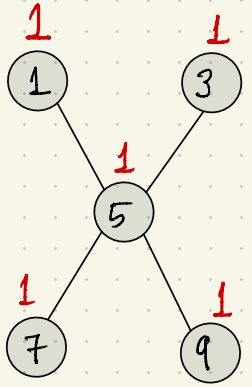
for i = 1 to n // try all vertices

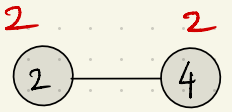
  if i is unexplored

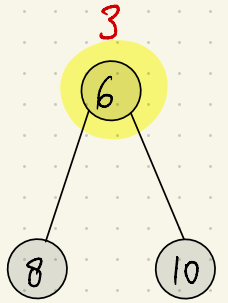
    increment CC count by 1

    BFS (G, i)

    (while updating CC number of a vertex extracted from queue







CONNECTED COMPONENTS via BFS

mark all vertices 1, 2, ..., n as unexplored

initialize CC count := 0

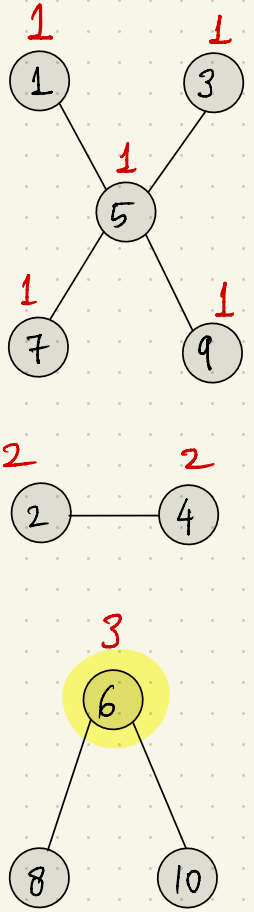
for i = 1 to n // try all vertices

  if i is unexplored

    increment CC count by 1

    BFS (G, i)

      while updating CC number of a vertex extracted from queue



CONNECTED COMPONENTS via BFS

mark all vertices $1, 2, \ldots, n$ as unexplored

initialize CC count := 0

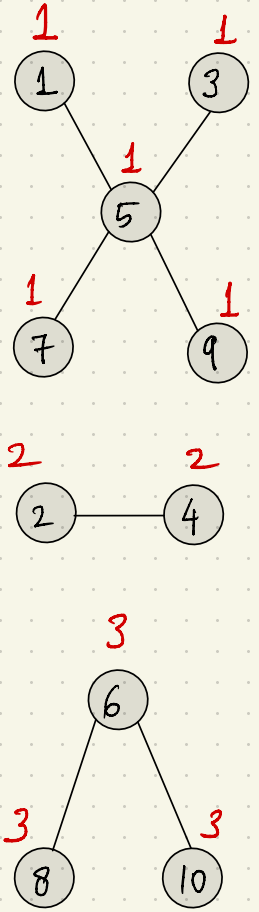
for i = 1 to n // try all vertices

  if i is unexplored

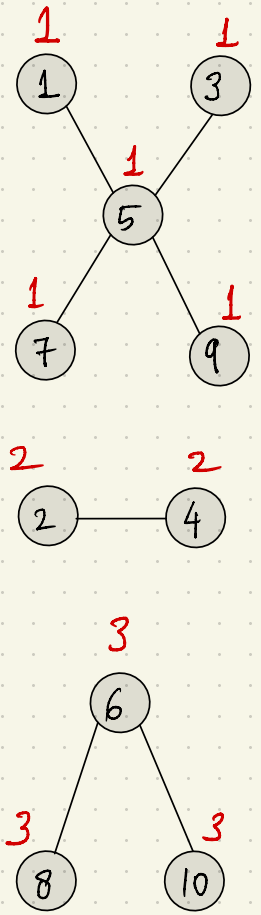
    increment CC count by 1

    BFS (G, i)

    (while updating CC number of a vertex extracted from queue



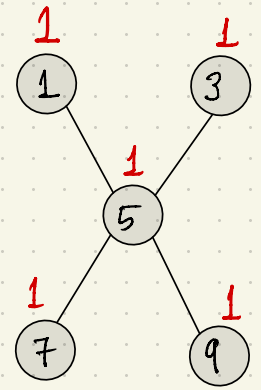
CONNECTED COMPONENTS via BFS

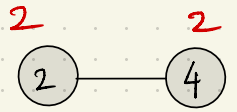


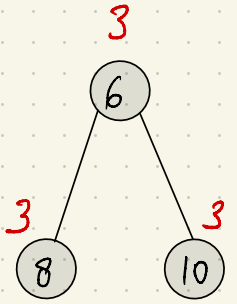
CONNECTED COMPONENTS via BFS

Claim: (a) At termination,

$cc[u] = cc[v] \iff u$ and v in same connected component





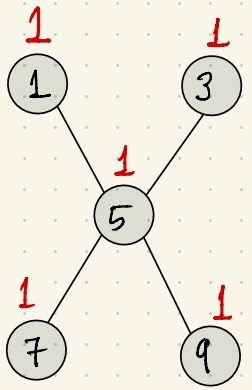


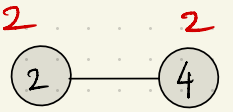
CONNECTED COMPONENTS via BFS

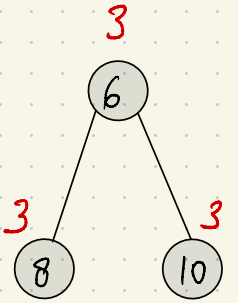
Claim: (a) At termination,

$cc[u] = cc[v] \iff u$ and v in same connected component

Idea: correctness of BFS + explored vertex is never marked unexplored







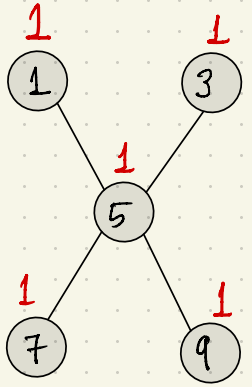
CONNECTED COMPONENTS via BFS

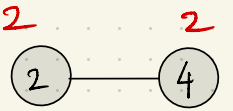
Claim: (a) At termination,

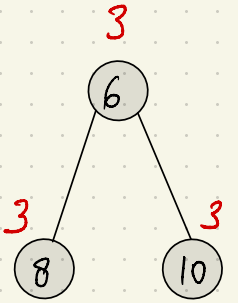
$cc[u] = cc[v] \iff u$ and v in same connected component

Idea: correctness of BFS + explored vertex is never marked unexplored

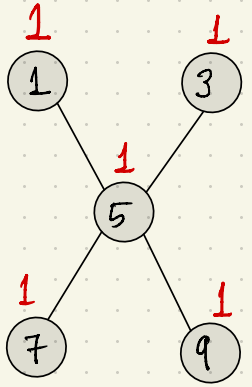
(b) Algo runs in $O(m+n)$ time







CONNECTED COMPONENTS via BFS

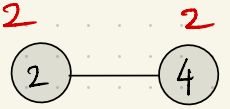


Claim: (a) At termination,

$cc[u] = cc[v] \iff u$ and v in same connected component

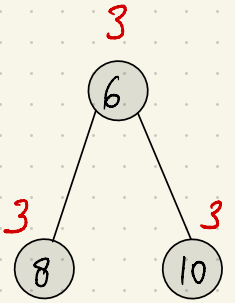
Idea: Correctness of BFS +

explored vertex is never marked unexplored

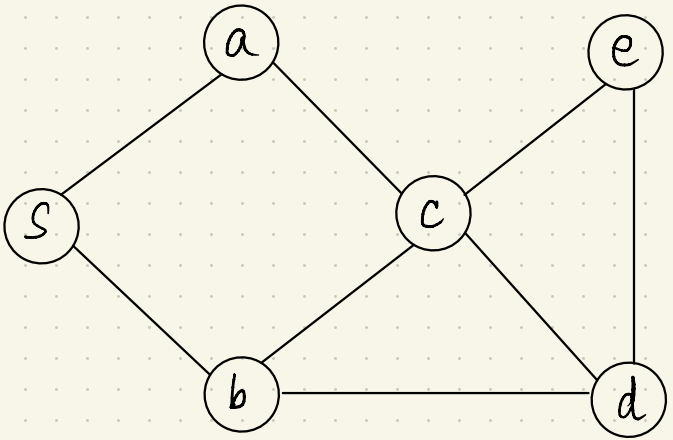


(b) Algo runs in $O(m+n)$ time

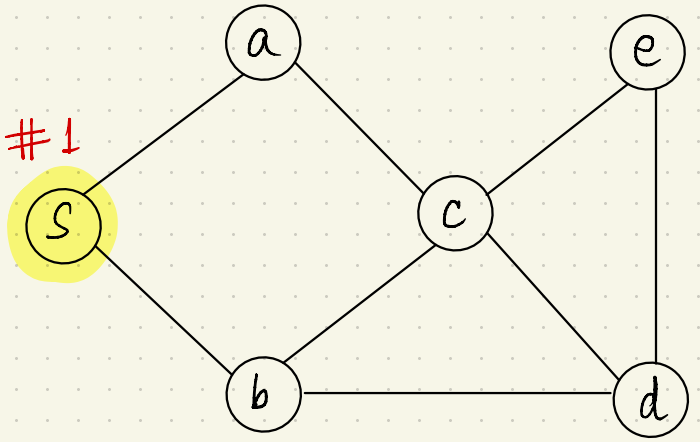
Idea: Time per component = $O(m_s + n_s)$.



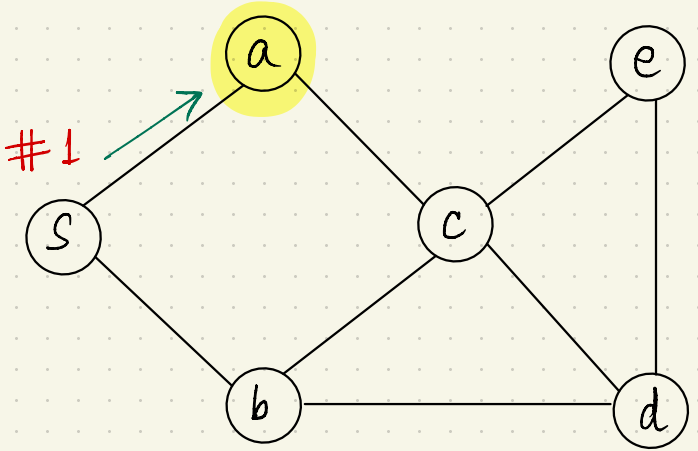
DEPTH-FIRST SEARCH



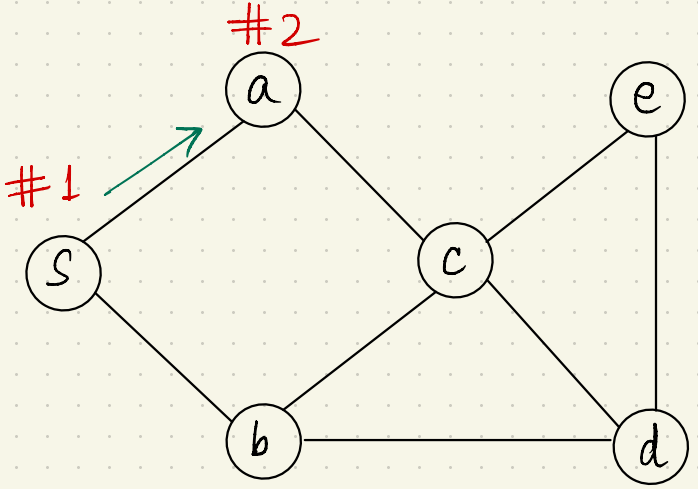
DEPTH-FIRST SEARCH



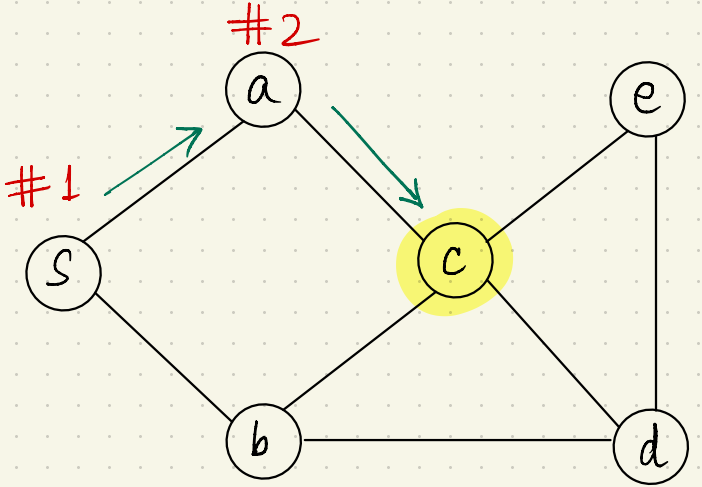
DEPTH-FIRST SEARCH



DEPTH-FIRST SEARCH

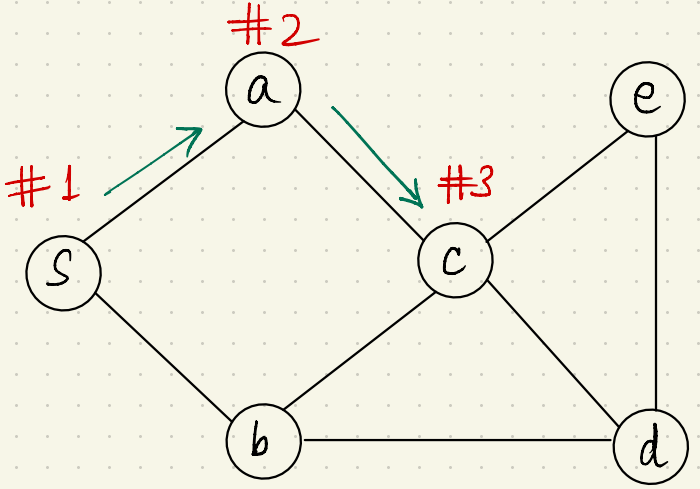


DEPTH-FIRST SEARCH



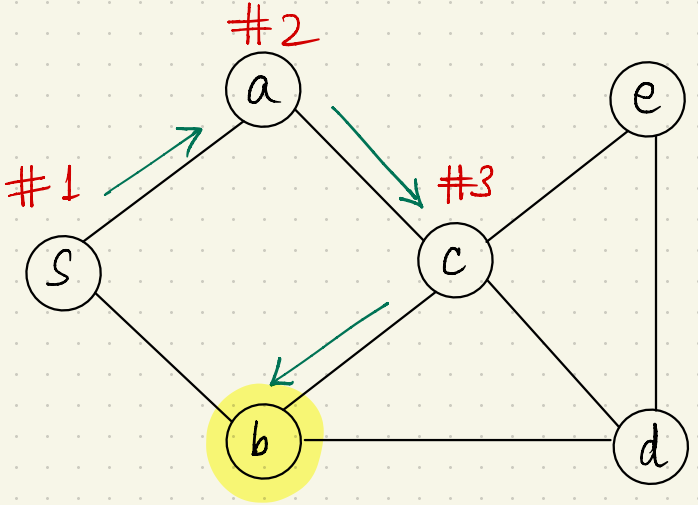
Differs from BFS !

DEPTH-FIRST SEARCH

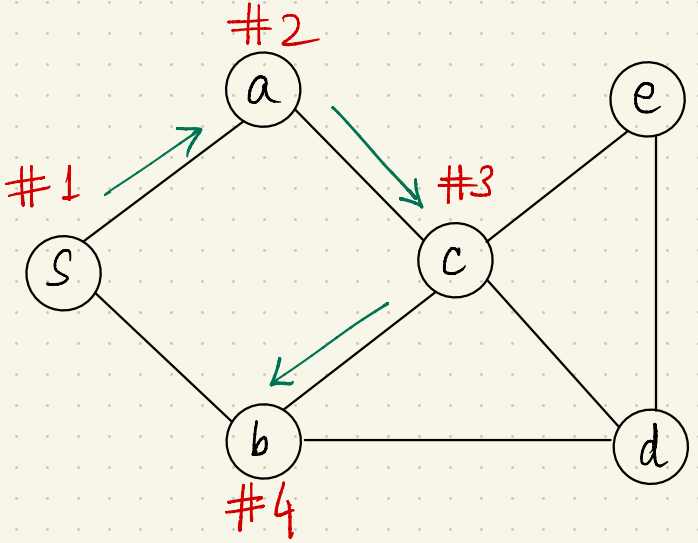


Differs from BFS !

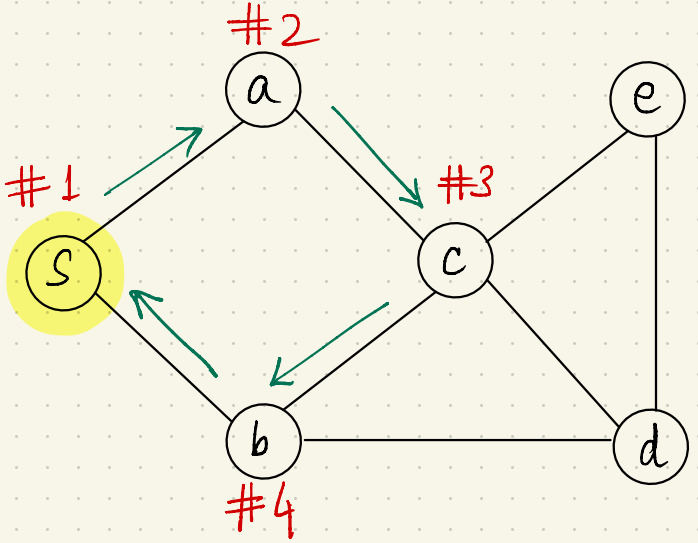
DEPTH-FIRST SEARCH



DEPTH-FIRST SEARCH

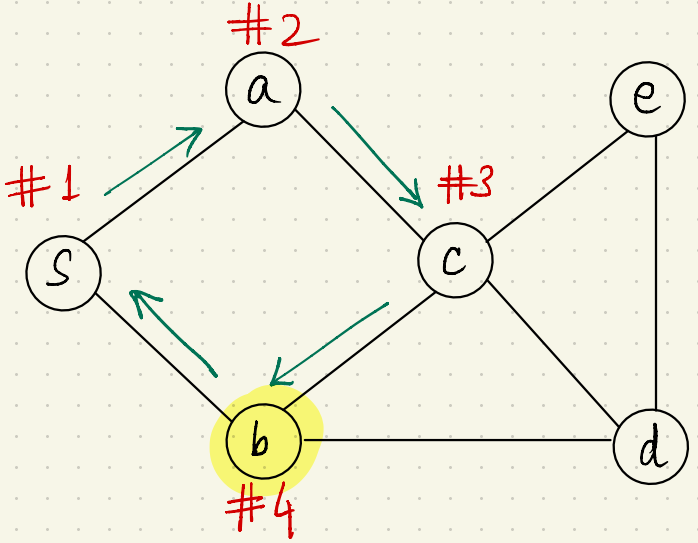


DEPTH-FIRST SEARCH



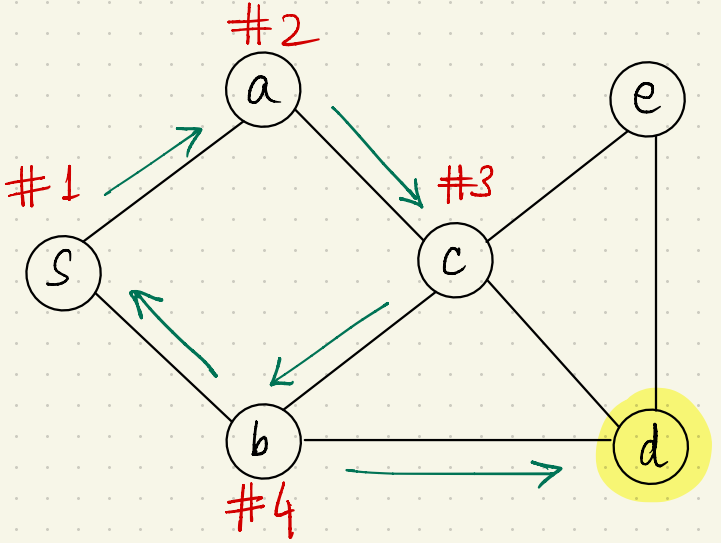
s is already explored ⟹ backtrack!

DEPTH-FIRST SEARCH

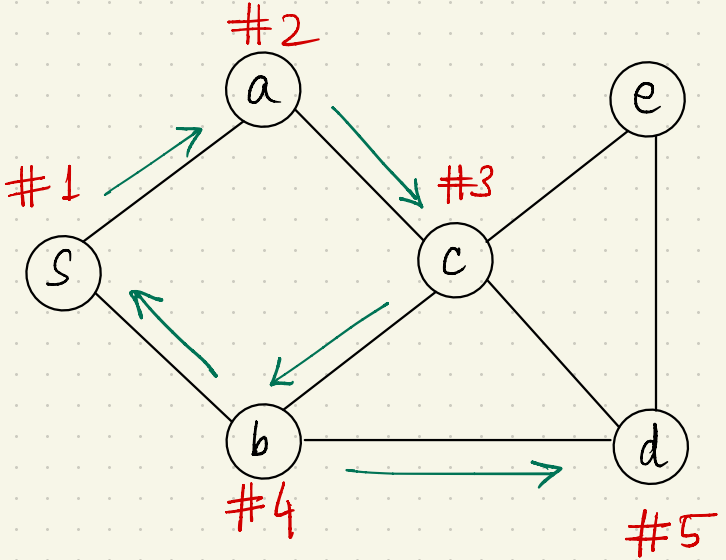


s is already explored $\Rightarrow$ backtrack!

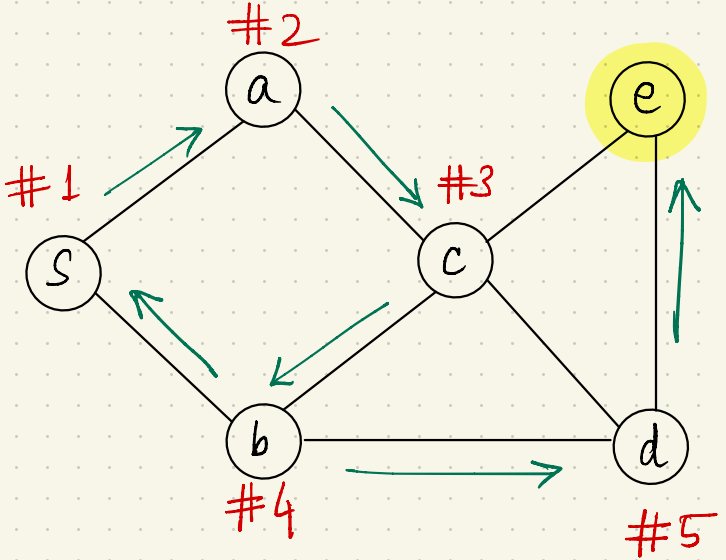
DEPTH-FIRST SEARCH



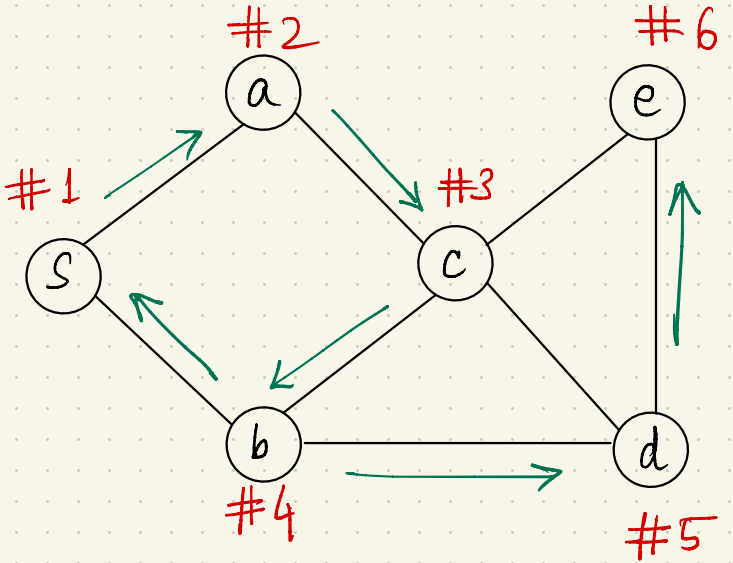
DEPTH-FIRST SEARCH



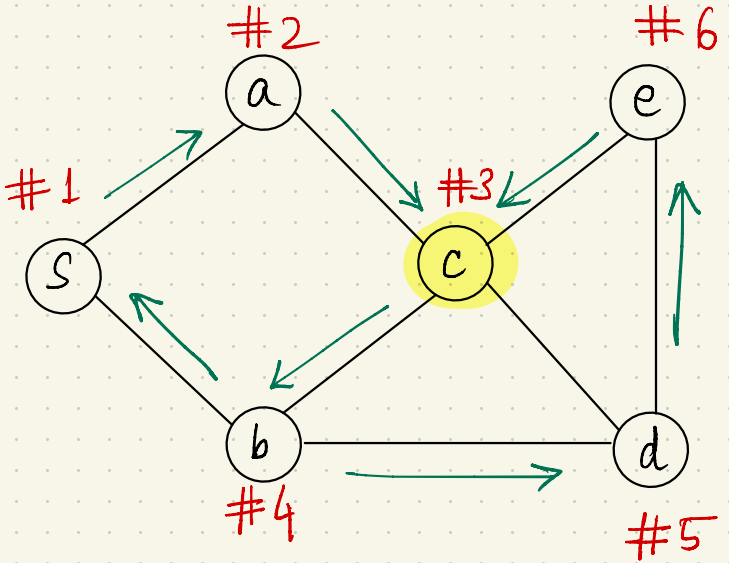
DEPTH-FIRST SEARCH



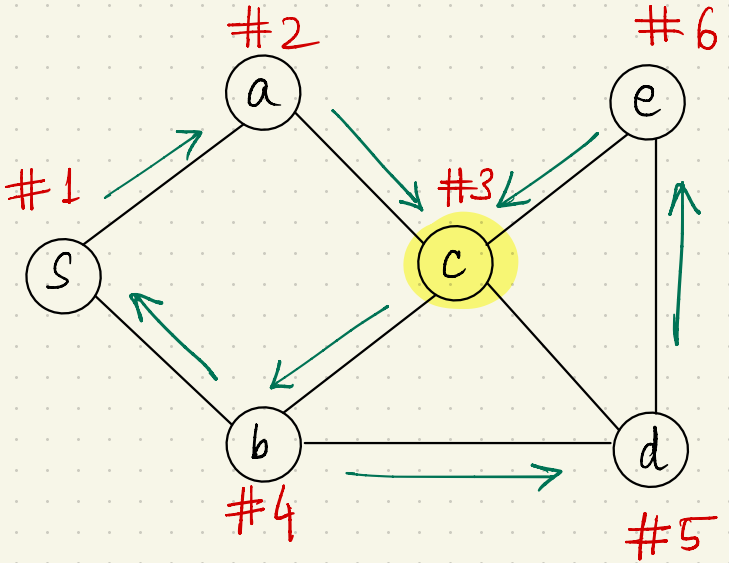
DEPTH-FIRST SEARCH



DEPTH-FIRST SEARCH

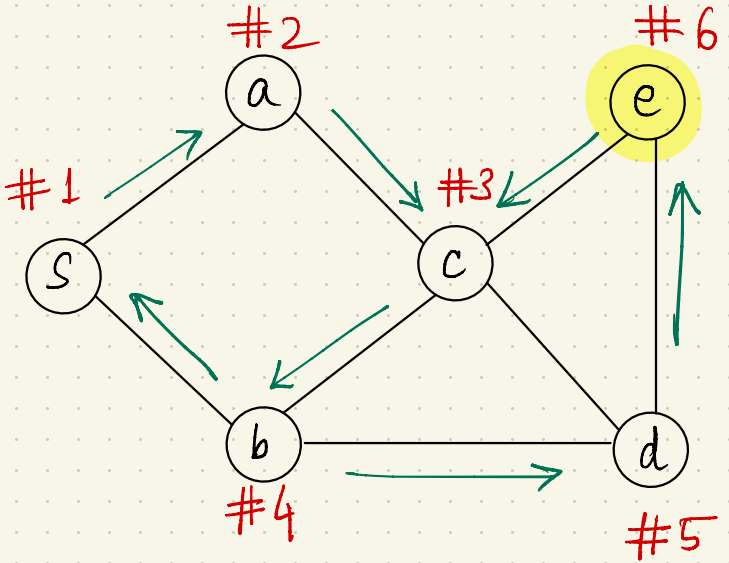


DEPTH-FIRST SEARCH



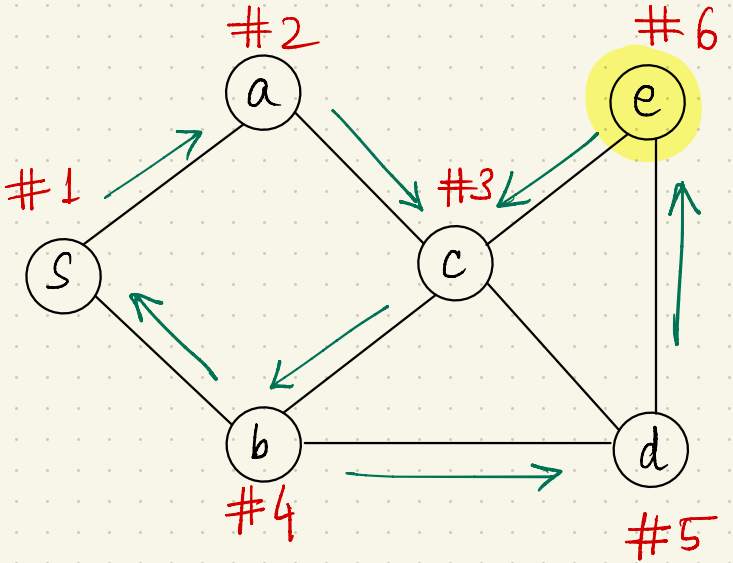
c is already explored $\Rightarrow$ backtrack!

DEPTH-FIRST SEARCH



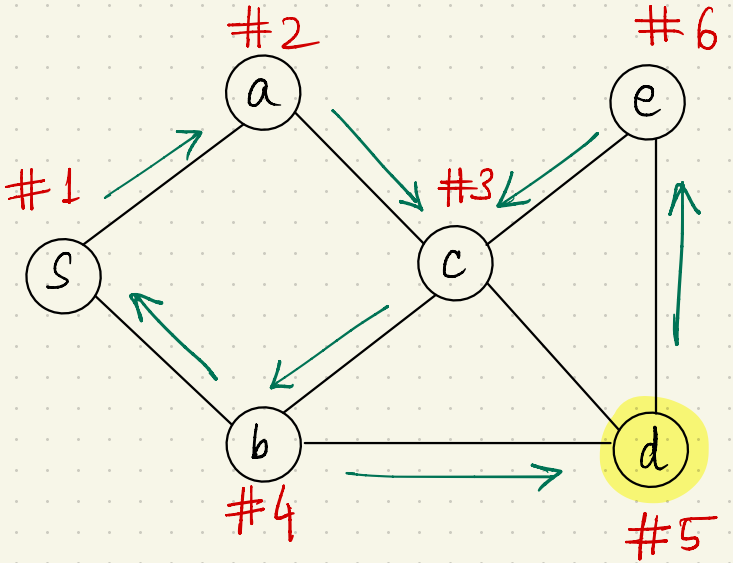
c is already explored ⟹ backtrack!

DEPTH-FIRST SEARCH



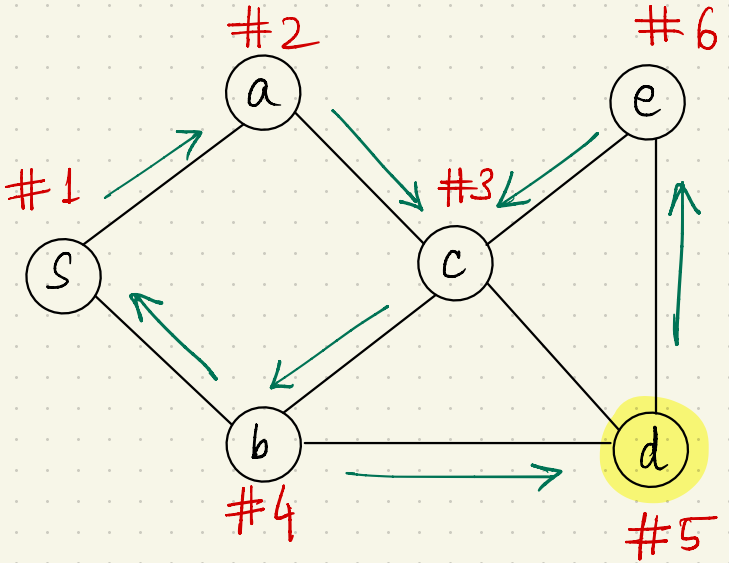
No unexplored neighbors of e ⇒ backtrack!

DEPTH-FIRST SEARCH

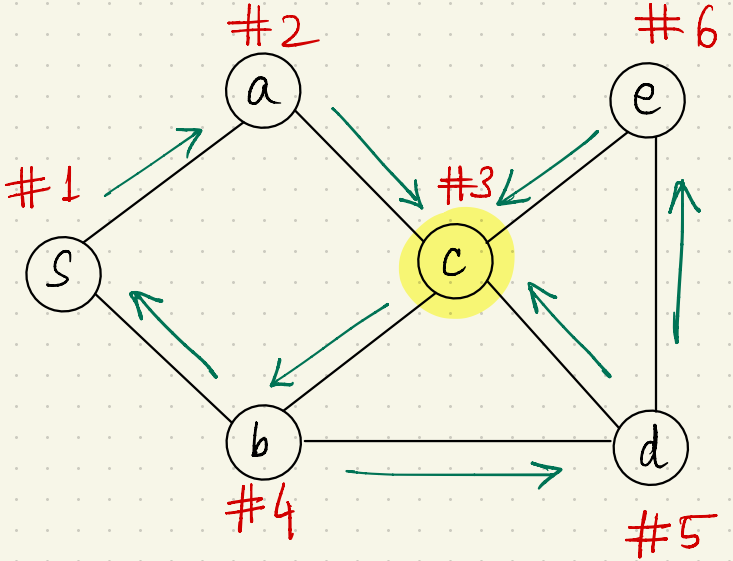


No unexplored neighbors of e ⇒ backtrack!

DEPTH-FIRST SEARCH

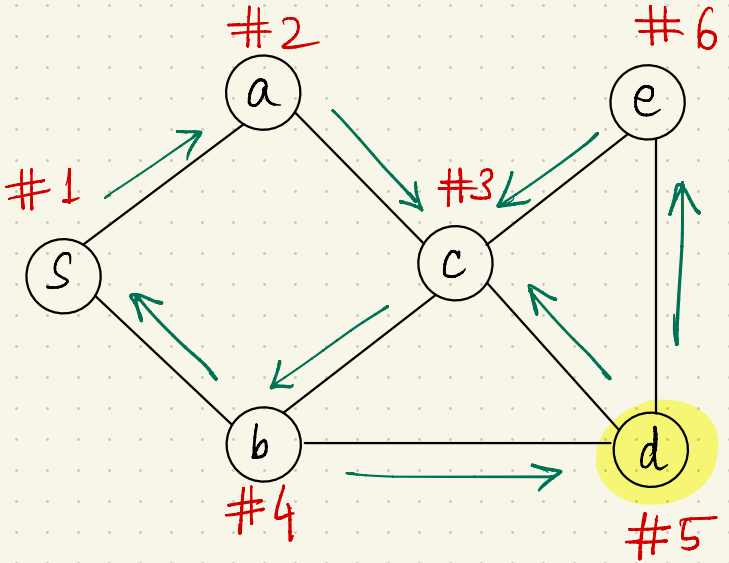


DEPTH-FIRST SEARCH



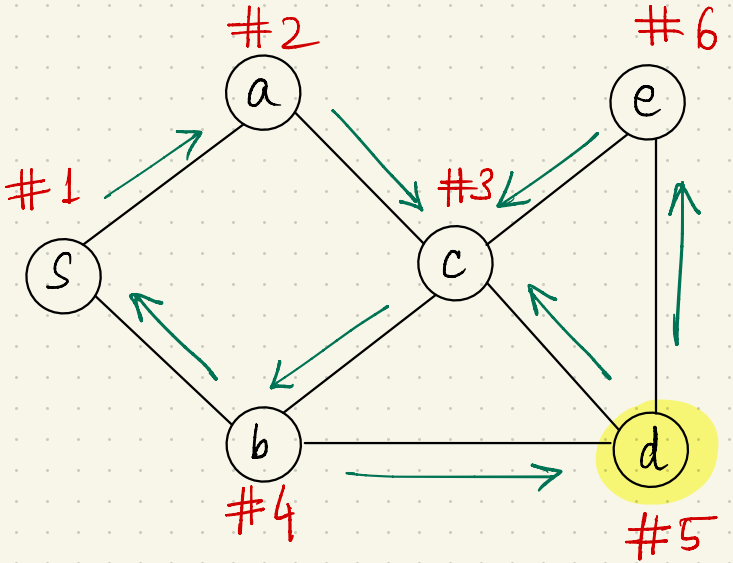
c is already explored ⟹ backtrack!

DEPTH-FIRST SEARCH



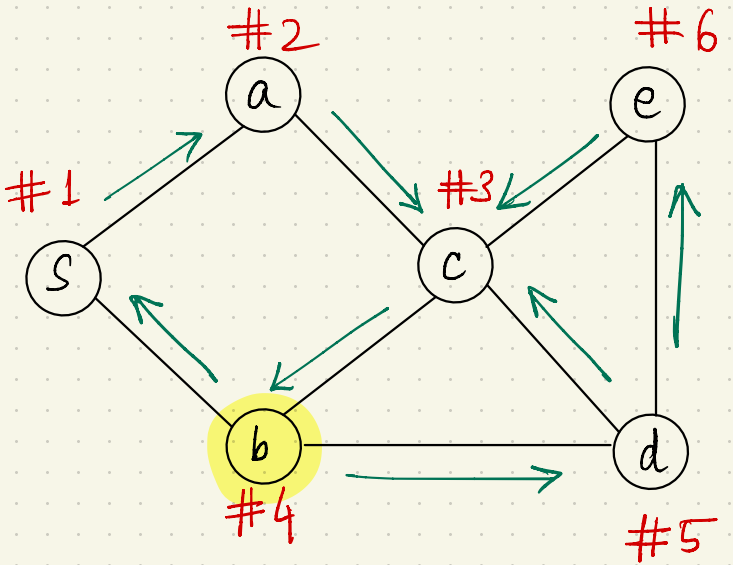
c is already explored ⟹ backtrack!

DEPTH-FIRST SEARCH



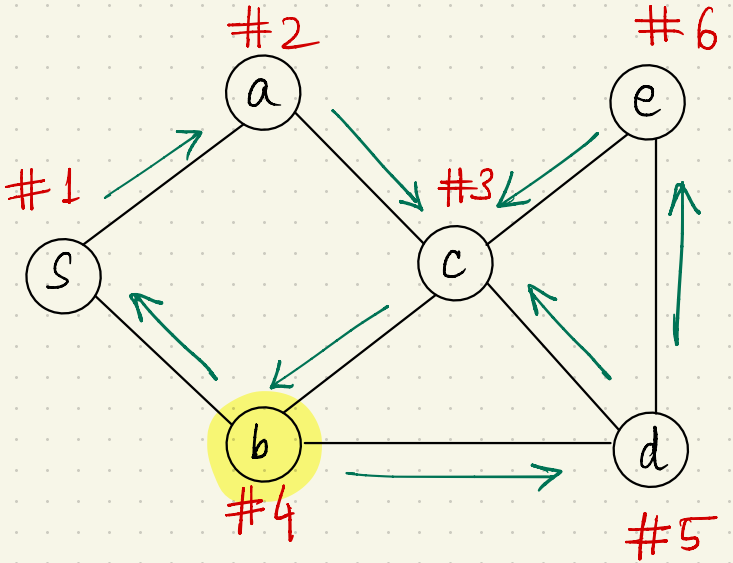
No unexplored neighbors of d ⇒ backtrack!

DEPTH-FIRST SEARCH



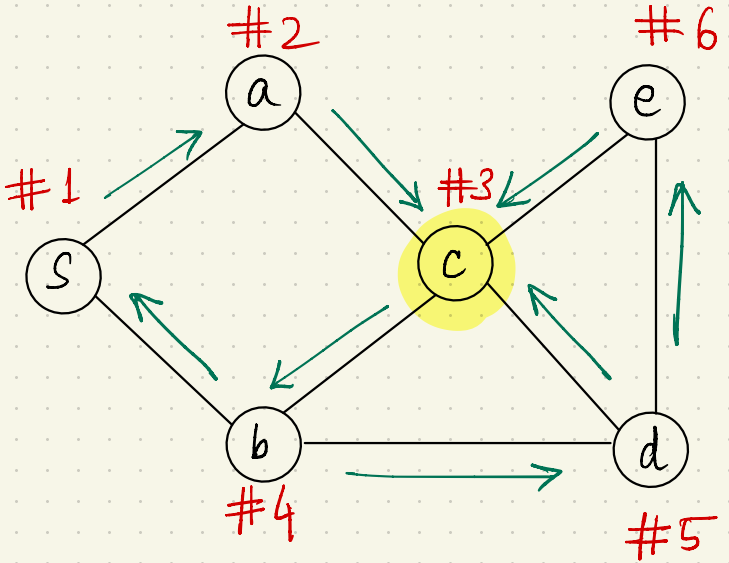
No unexplored neighbors of d ⇒ backtrack!

DEPTH-FIRST SEARCH



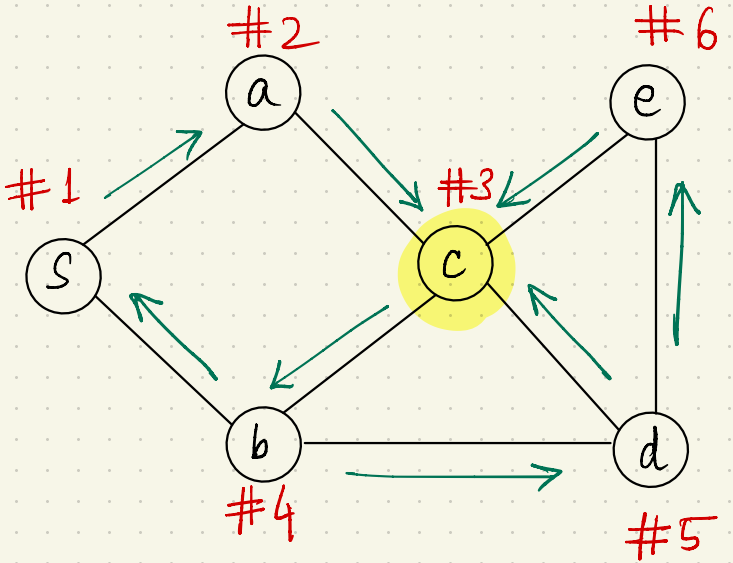
No unexplored neighbors of b ⇒ backtrack!

DEPTH-FIRST SEARCH



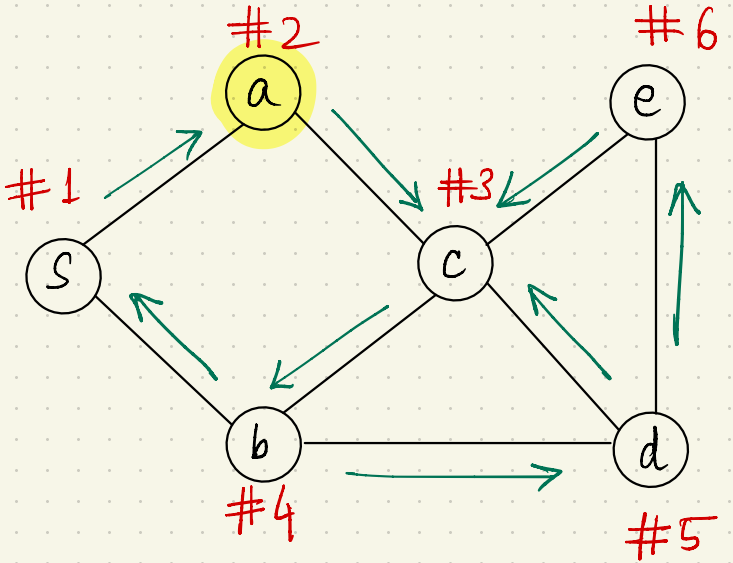
No unexplored neighbors of b ⇒ backtrack!

DEPTH-FIRST SEARCH



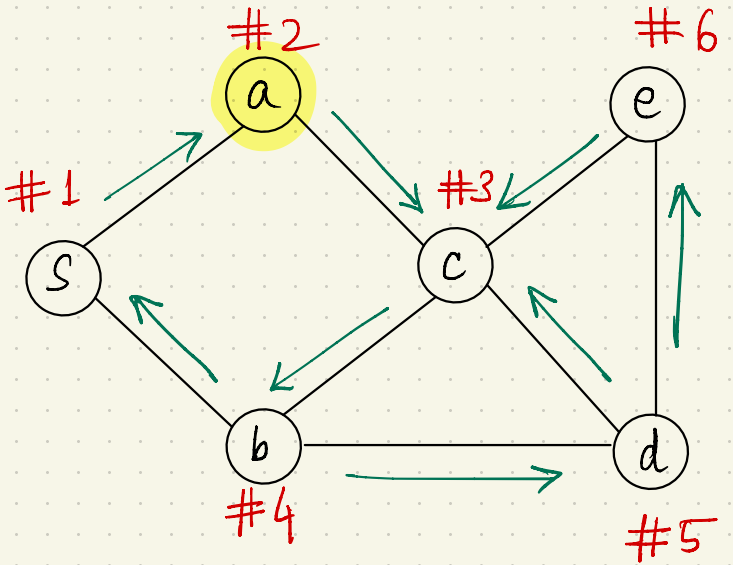
No unexplored neighbors of c ⇒ backtrack!

DEPTH-FIRST SEARCH



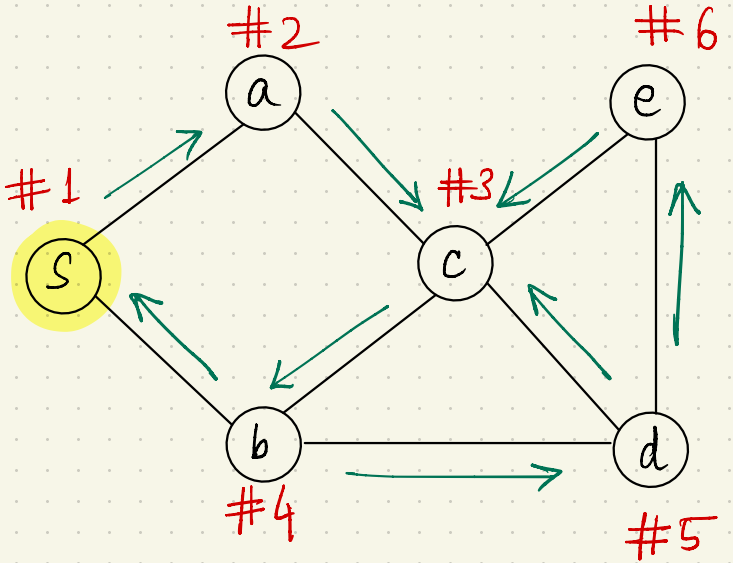
No unexplored neighbors of c ⇒ backtrack!

DEPTH-FIRST SEARCH



No unexplored neighbors of a ⇒ backtrack!

DEPTH-FIRST SEARCH



Back home!

DEPTH-FIRST SEARCH

Iterative version

DEPTH-FIRST SEARCH

Iterative version

mark all vertices as unexplored

S := a stack data structure (LIFO), initialized with s

while $S \neq \emptyset$

- remove the top node of S, say v ("pop")
- if v is unexplored
  - mark v as explored
  - for each (v, w) in adj. list of v
    - add w to the front of S ("push")

DEPTH-FIRST SEARCH

Iterative version

mark all vertices as unexplored

S := a stack data structure (LIFO), initialized with s

while $S \neq \emptyset$

- remove the top node of S, say v ("pop")
- if v is unexplored
  - mark v as explored
  - for each (v, w) in adj. list of v
    - add w to the front of S ("push")

DEPTH-FIRST SEARCH

Recursive version

DEPTH-FIRST SEARCH

Recursive version

```
DFS (G, s)          // all vertices unexplored before the call

    mark s as explored

    for each (s, v) in adj. list of s
    |       if v is unexplored
    |       |     DFS(G, v)
```

DEPTH-FIRST SEARCH

Claim 1: At the end of DFS

v is explored $\iff$ $\exists$ path from s to v in G

DEPTH-FIRST SEARCH

Claim 1: At the end of DFS

v is explored $\iff$ $\exists$ path from s to v in G

Reason : DFS is a special case of Generic Algorithm.

DEPTH-FIRST SEARCH

Claim 1: At the end of DFS

v is explored $\iff$ $\exists$ path from s to v in G

Reason: DFS is a special case of Generic Algorithm.

Claim 2: Running time is $O(n_s + m_s)$.

n_s: # reachable vertices

m_s: # reachable edges

DEPTH-FIRST SEARCH

Claim 1: At the end of DFS

v is explored $\iff$ $\exists$ path from s to v in G

Reason: DFS is a special case of Generic Algorithm.

Claim 2: Running time is $O(n_s + m_s)$.

n_s: # reachable vertices

m_s: # reachable edges

Corollary: DFS can compute connected components in linear time.

APPLICATIONS OF DFS

Topological ordering

Strongly Connected Components

APPLICATIONS OF DFS

Topological ordering

Strongly Connected Components

TOPOLOGICAL ORDERING

Directed graph $G = (V, E)$

TOPOLOGICAL ORDERING

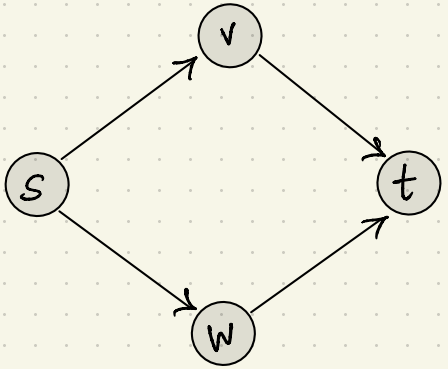
Directed graph $G = (V, E)$

A labeling f of G's vertices such that:

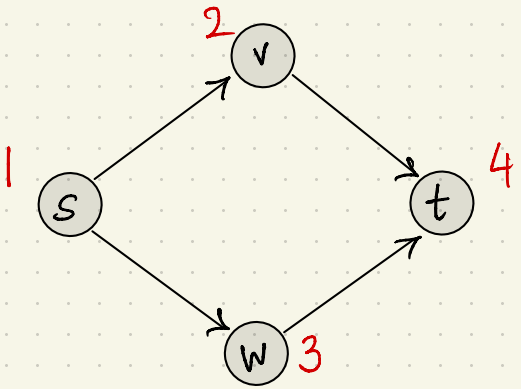
* unique $f(v) \in \{1, 2, \ldots, n\}$ for every $v \in V$

* for every $(v, w) \in E$ $\quad f(v) < f(w)$.

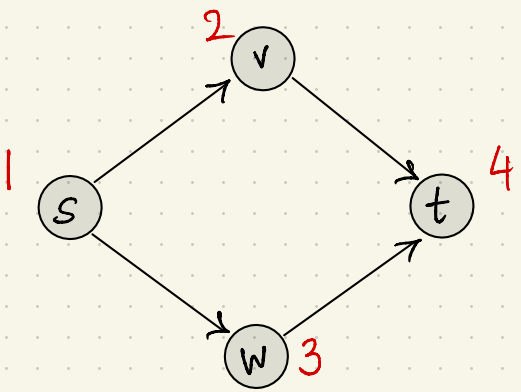
TOPOLOGICAL ORDERING

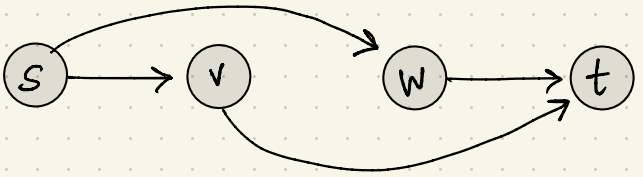


TOPOLOGICAL ORDERING

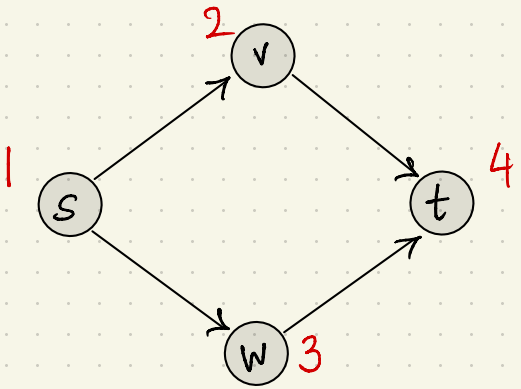


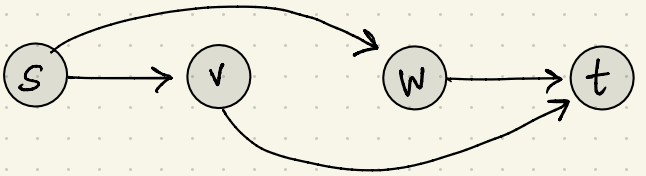
TOPOLOGICAL ORDERING

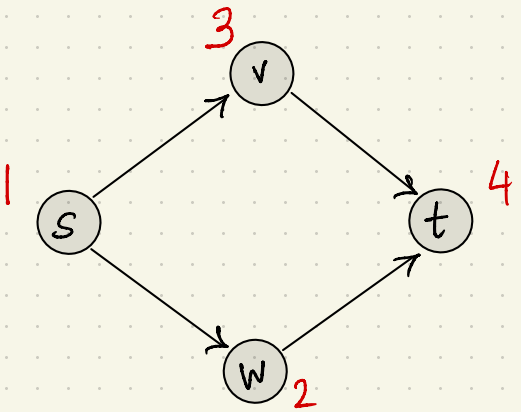


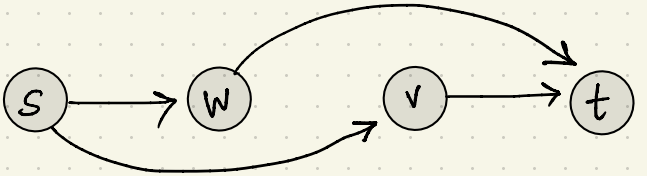


TOPOLOGICAL ORDERING

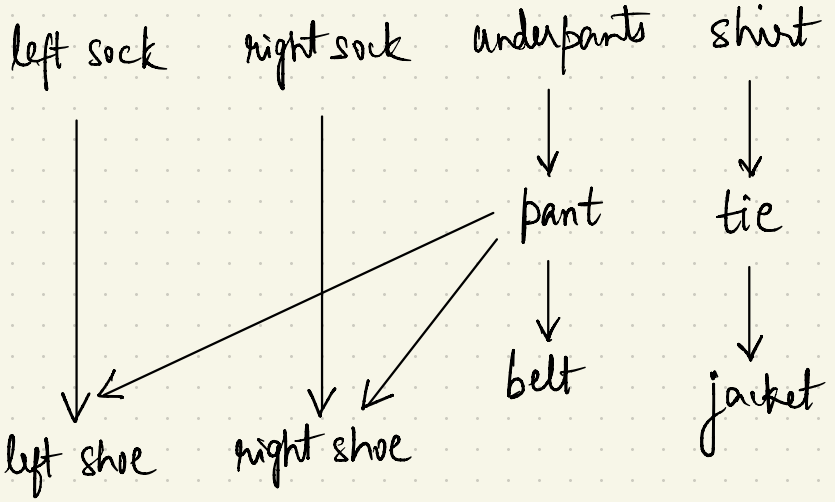




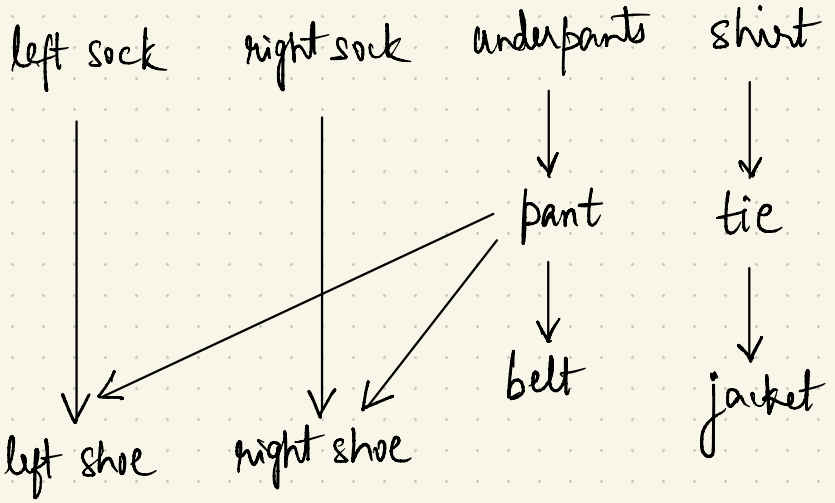




TOPOLOGICAL ORDERING

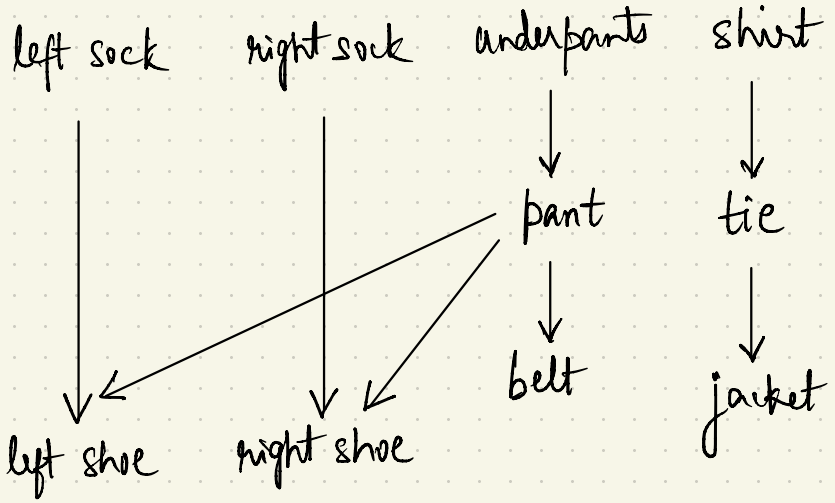


TOPOLOGICAL ORDERING



underpants
shirt
pant
belt
tie
jacket
left sock
right sock
left shoe
right shoe

TOPOLOGICAL ORDERING



underpants	left sock
shirt	shirt
pant	tie
belt	underpants
tie	right sock
jacket	pants
left sock	right shoe
right sock	belt
left shoe	jacket
right shoe	left shoe

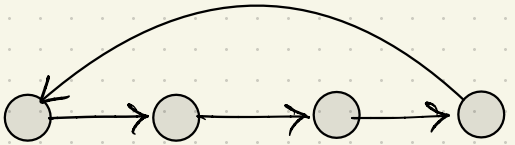
TOPOLOGICAL ORDERING

When does a directed graph have a topological ordering?

TOPOLOGICAL ORDERING

When does a directed graph have a topological ordering?

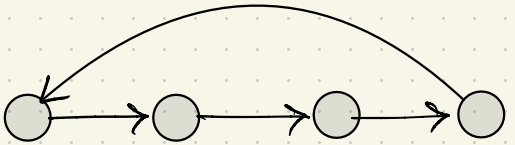
Necessary : No cycle



TOPOLOGICAL ORDERING

When does a directed graph have a topological ordering?

Necessary: No cycle



Also sufficient!

TOPOLOGICAL ORDERING

Theorem : Every directed acyclic graph has at least one topological ordering.

TOPOLOGICAL ORDERING

Theorem: Every directed acyclic graph has at least one topological ordering.

Lemma: Every directed acyclic graph has at least one sink.

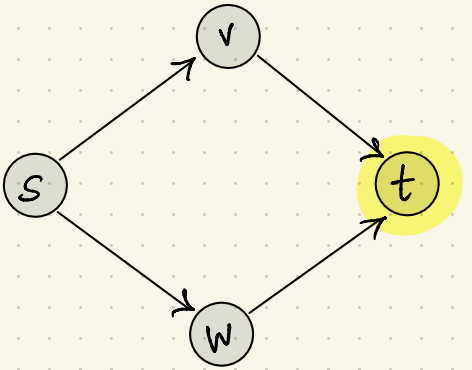
a vertex with no outgoing edges

TOPOLOGICAL ORDERING

Theorem: Every directed acyclic graph has at least one topological ordering.

Lemma: Every directed acyclic graph has at least one sink.

a vertex with no outgoing edges



TOPOLOGICAL ORDERING

Theorem: Every directed acyclic graph has at least one topological ordering.

Lemma: Every directed acyclic graph has at least one sink.

a vertex with no outgoing edges

Proof of Lemma: Suppose not. Then, every vertex has some outgoing edge

"Following your nose" will give a cycle. Contradiction!

TOPOLOGICAL ORDERING

Theorem: Every directed acyclic graph has at least one topological ordering.

Lemma: Every directed acyclic graph has at least one sink.

a vertex with no outgoing edges

Lemma: Every directed acyclic graph has at least one source.

a vertex with no incoming edges

TOPOLOGICAL ORDERING

Theorem: Every directed acyclic graph has at least one topological ordering.

Proof: Assign $f[v] = n$ to sink vertex v (exists!)

TOPOLOGICAL ORDERING

Theorem: Every directed acyclic graph has at least one topological ordering.

Proof: Assign $f[v] = n$ to sink vertex v (exists!)

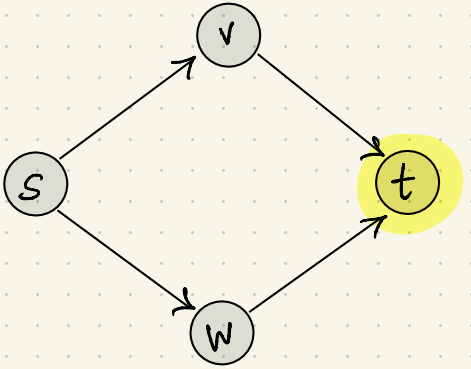
Recurse on $G \setminus \{v\}$ $\longrightarrow$ must be directed acyclic. ∎

TOPOLOGICAL ORDERING

Theorem: Every directed acyclic graph has at least one topological ordering.

Proof: Assign $f[v] = n$ to sink vertex v (exists!)

Recurse on $G \setminus \{v\}$ $\longrightarrow$ must be directed acyclic. ∎

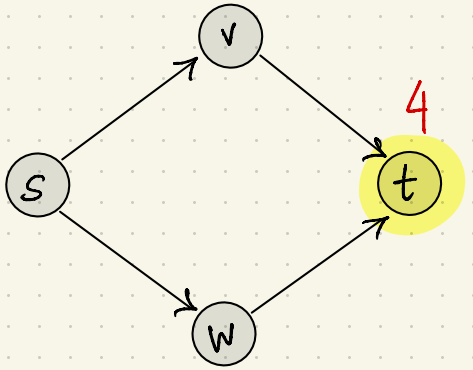


TOPOLOGICAL ORDERING

Theorem: Every directed acyclic graph has at least one topological ordering.

Proof: Assign $f[v] = n$ to sink vertex v (exists!)

Recurse on $G \setminus \{v\}$ $\longrightarrow$ must be directed acyclic. ∎

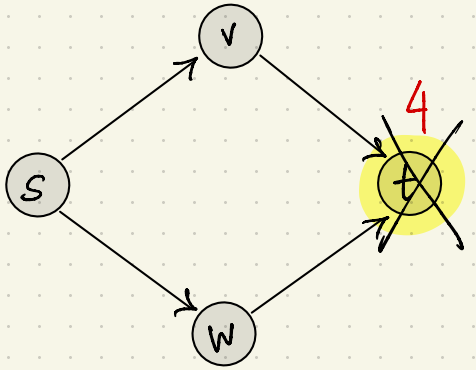


TOPOLOGICAL ORDERING

Theorem: Every directed acyclic graph has at least one topological ordering.

Proof: Assign $f[v] = n$ to sink vertex v (exists!)

Recurse on $G \setminus \{v\}$ $\longrightarrow$ must be directed acyclic. ∎

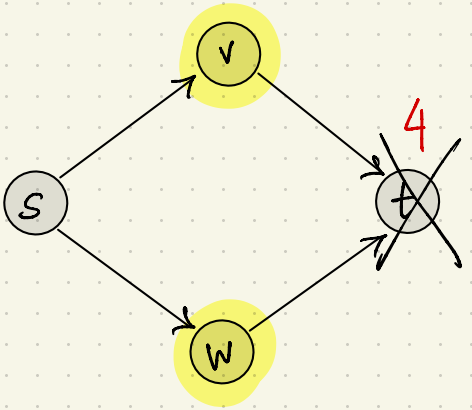


TOPOLOGICAL ORDERING

Theorem: Every directed acyclic graph has at least one topological ordering.

Proof: Assign $f[v] = n$ to sink vertex v (exists!)

Recurse on $G \setminus \{v\}$ $\longrightarrow$ must be directed acyclic. ∎

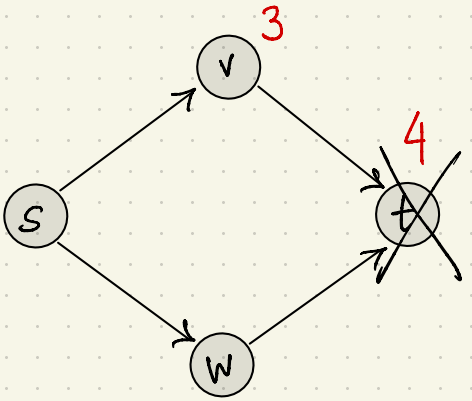


TOPOLOGICAL ORDERING

Theorem: Every directed acyclic graph has at least one topological ordering.

Proof: Assign $f[v] = n$ to sink vertex v (exists!)

Recurse on $G \setminus \{v\}$ $\longrightarrow$ must be directed acyclic. ∎

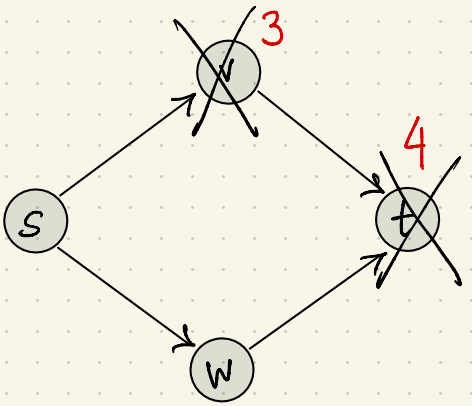


TOPOLOGICAL ORDERING

Theorem: Every directed acyclic graph has at least one topological ordering.

Proof: Assign $f[v] = n$ to sink vertex v (exists!)

Recurse on $G \setminus \{v\}$ $\longrightarrow$ must be directed acyclic. ∎

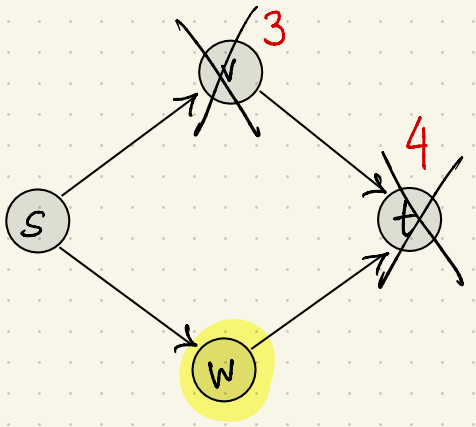


TOPOLOGICAL ORDERING

Theorem: Every directed acyclic graph has at least one topological ordering.

Proof: Assign $f[v] = n$ to sink vertex v (exists!)

Recurse on $G \setminus \{v\}$ $\longrightarrow$ must be directed acyclic. ∎

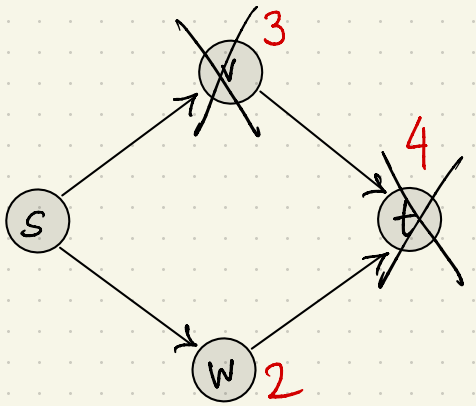


TOPOLOGICAL ORDERING

Theorem: Every directed acyclic graph has at least one topological ordering.

Proof: Assign $f[v] = n$ to sink vertex v (exists!)

Recurse on $G \setminus \{v\}$ $\longrightarrow$ must be directed acyclic. ∎

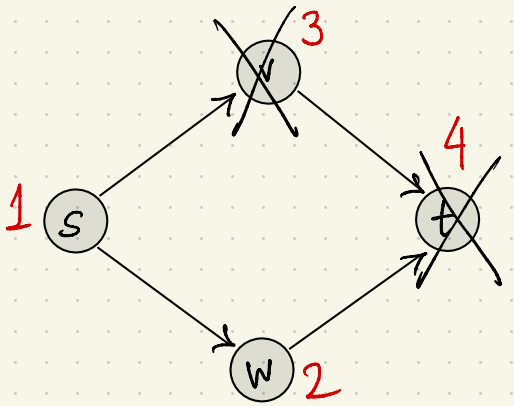


TOPOLOGICAL ORDERING

Theorem: Every directed acyclic graph has at least one topological ordering.

Proof: Assign $f[v] = n$ to sink vertex v (exists!)

Recurse on $G \setminus \{v\}$ $\longrightarrow$ must be directed acyclic. ∎

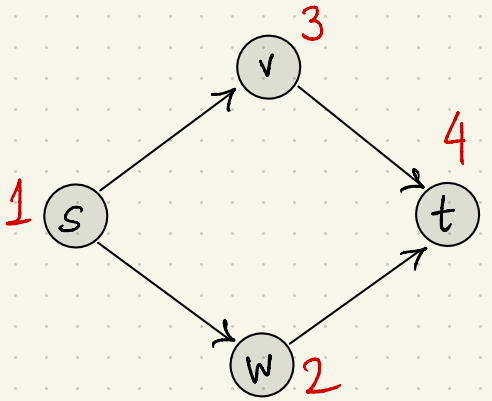


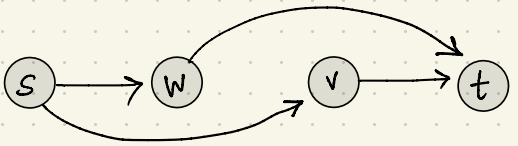
TOPOLOGICAL ORDERING

Theorem: Every directed acyclic graph has at least one topological ordering.

Proof: Assign $f[v] = n$ to sink vertex v (exists!)

Recurse on $G \setminus \{v\}$ $\longrightarrow$ must be directed acyclic. ▨





TOPOLOGICAL ORDERING

Theorem: Every directed acyclic graph has at least one topological ordering.

Algorithm

1. Find a sink vertex v.
2. Assign to it the largest available label and recurse on $G \setminus \{v\}$.

TOPOLOGICAL ORDERING

Theorem: Every directed acyclic graph has at least one topological ordering.

Algorithm

1. Find a sink vertex v.
2. Assign to it the largest available label and recurse on $G \setminus \{v\}$.

Correctness

If $f[v] = i$, no edges to vertices with $f[v] < i$.

TOPOLOGICAL ORDERING

Theorem: Every directed acyclic graph has at least one topological ordering.

Algorithm

1. Find a sink vertex v.
2. Assign to it the largest available label and recurse on $G \setminus \{v\}$.

Running time

TOPOLOGICAL ORDERING

Theorem: Every directed acyclic graph has at least one topological ordering.

Algorithm

1. Find a sink vertex v. $\longrightarrow O(n)$
2. Assign to it the largest available label and recurse on $G \setminus \{v\}$.

Running time

$O(n^2)$

TOPOLOGICAL ORDERING

Theorem: Every directed acyclic graph has at least one topological ordering.

Algorithm

1. Find a sink vertex v. $\longrightarrow O(n)$
2. Assign to it the largest available label and recurse on $G \setminus \{v\}$.

Running time

$O(n^2)$

Can we do better for sparse graphs?

TOPOLOGICAL ORDERING via DFS

TOPOLOGICAL ORDERING via DFS

DFS pseudocode

```
DFS (G, s)        // all vertices unexplored before the call

    mark s as explored

    for each (s, v) in adj. list of s
        if v is unexplored
            DFS (G, v)
```

TOPOLOGICAL ORDERING via DFS

```
DFS (G, s)   // all vertices initially unexplored
mark s as explored
for each (s, v) in adj. list of s
    if v is unexplored
        DFS (G, v)
```

TOPOLOGICAL ORDERING via DFS

DFS-Loop (G)

DFS (G, s) // all vertices initially unexplored

mark s as explored

for each (s, v) in adj. list of s

    if v is unexplored

        DFS (G, v)

TOPOLOGICAL ORDERING via DFS

```
DFS-Loop(G)
mark all vertices unexplored
current_label := |V| // labeling f
```

```
DFS(G, s) // all vertices initially unexplored
mark s as explored
for each (s, v) in adj. list of s
    if v is unexplored
        DFS(G, v)
```

TOPOLOGICAL ORDERING via DFS

```
DFS-Loop(G)
mark all vertices unexplored
current_label := |V|  // labeling f
for each v ∈ V
    if v is unexplored
        DFS(G, v)
```

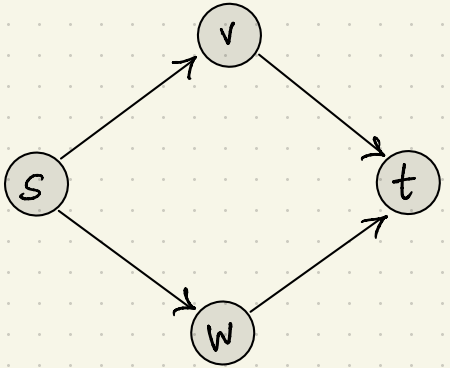
```
DFS(G, s)  // all vertices initially unexplored
mark s as explored
for each (s, v) in adj. list of s
    if v is unexplored
        DFS(G, v)
```

TOPOLOGICAL ORDERING via DFS

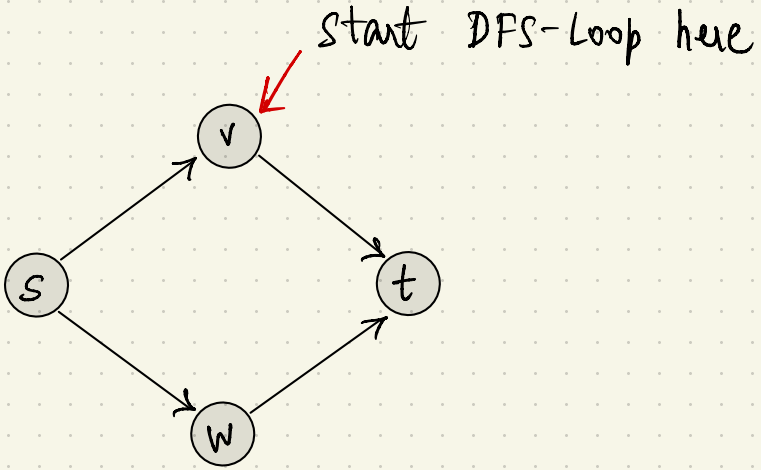
```
DFS-Loop(G)
mark all vertices unexplored
current_label := |V|  // labeling f
for each v ∈ V
    if v is unexplored
        DFS(G, v)
```

```
DFS(G, s)  // all vertices initially unexplored
mark s as explored
for each (s, v) in adj. list of s
    if v is unexplored
        DFS(G, v)
Set f[s] = current_label
decrease current_label by 1
```

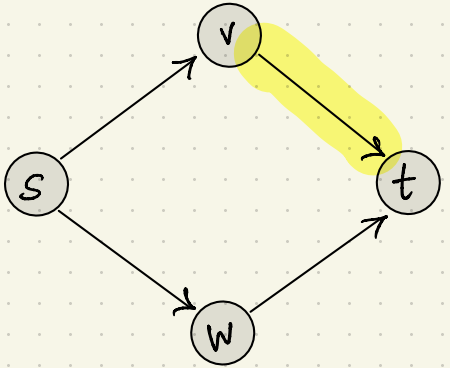
TOPOLOGICAL ORDERING via DFS



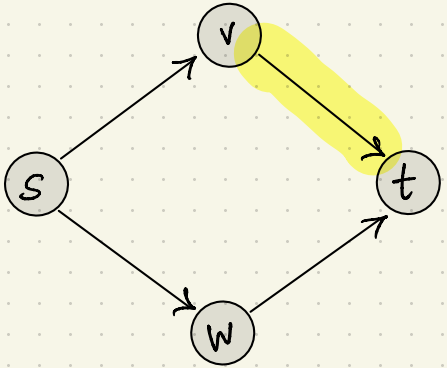
TOPOLOGICAL ORDERING via DFS



TOPOLOGICAL ORDERING via DFS

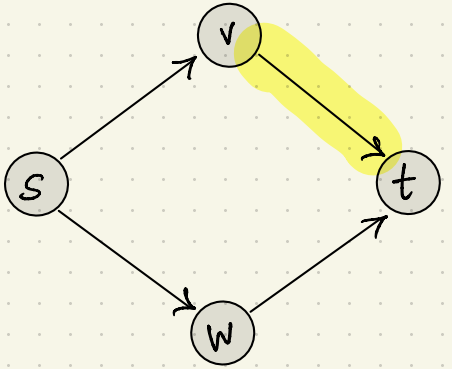


TOPOLOGICAL ORDERING via DFS



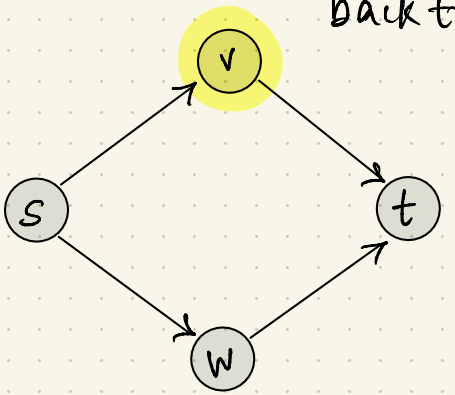
nowhere to go from t

TOPOLOGICAL ORDERING via DFS



$f[t] = 4$

TOPOLOGICAL ORDERING via DFS

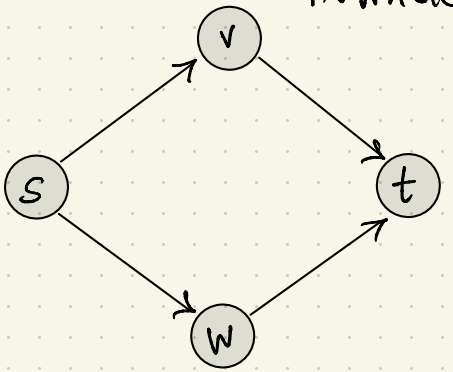


backtrack to v

$f[t] = 4$

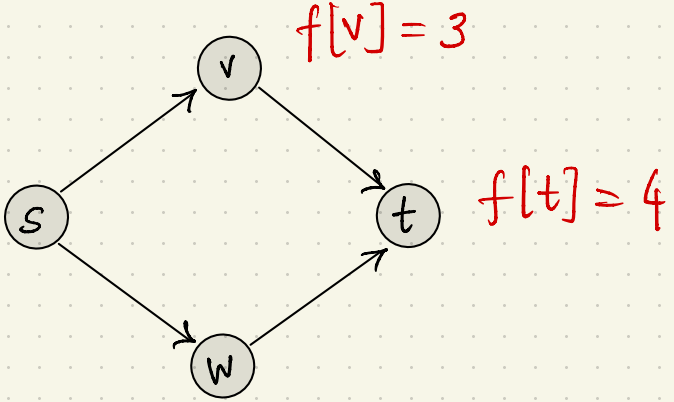
TOPOLOGICAL ORDERING via DFS

nowhere else to go from v

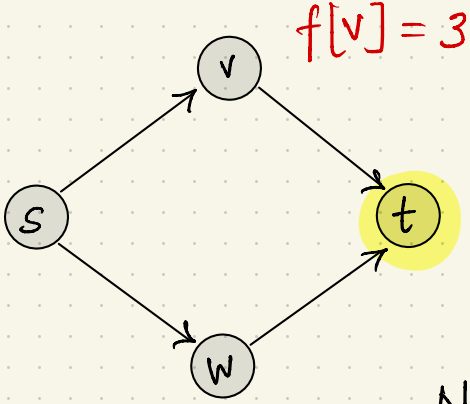


$f[t] = 4$

TOPOLOGICAL ORDERING via DFS



TOPOLOGICAL ORDERING via DFS



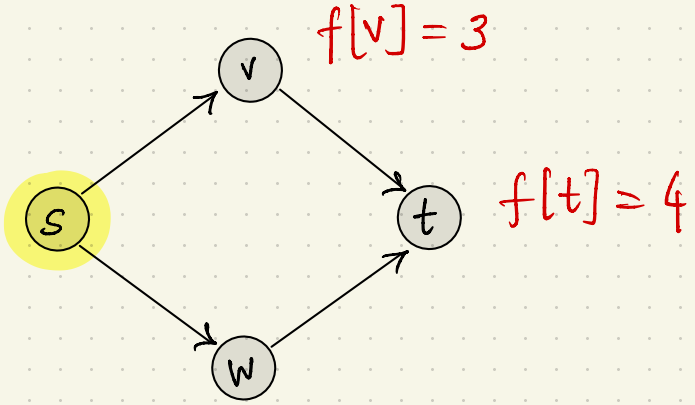
Next, DFS-loop considers t

but it is already explored

so skip it.

TOPOLOGICAL ORDERING via DFS

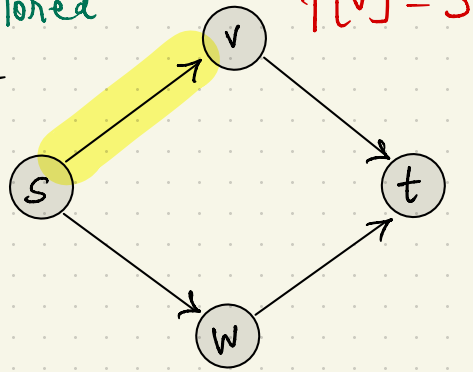
Next, DFS-Loop considers s.



TOPOLOGICAL ORDERING via DFS

v is explored

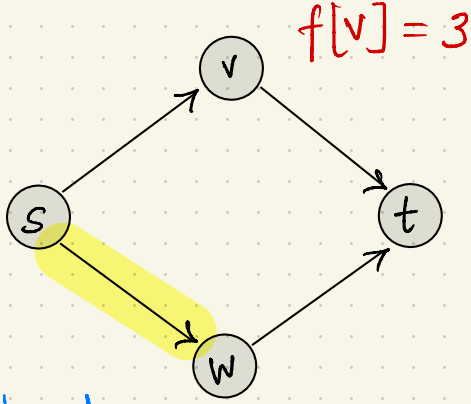
so skip it



f[v] = 3

f[t] = 4

TOPOLOGICAL ORDERING via DFS

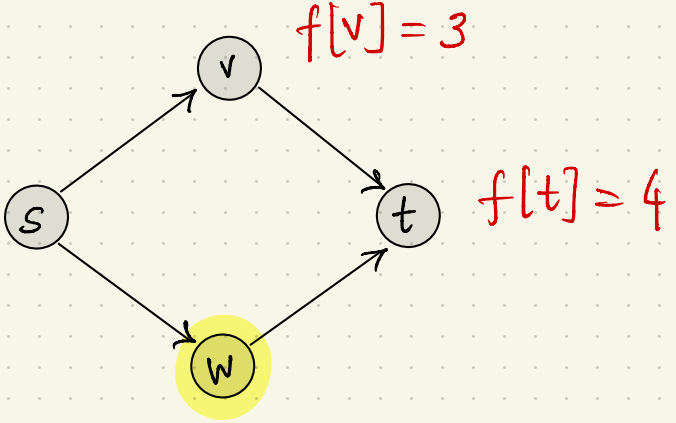


$f[v] = 3$

$f[t] = 4$

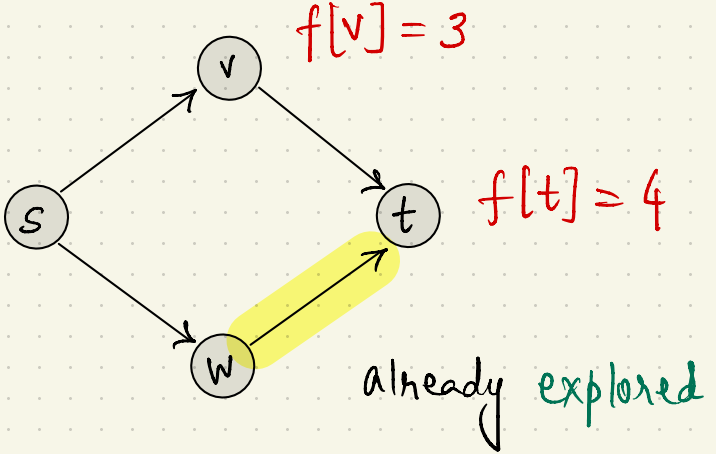
w is unexplored

TOPOLOGICAL ORDERING via DFS

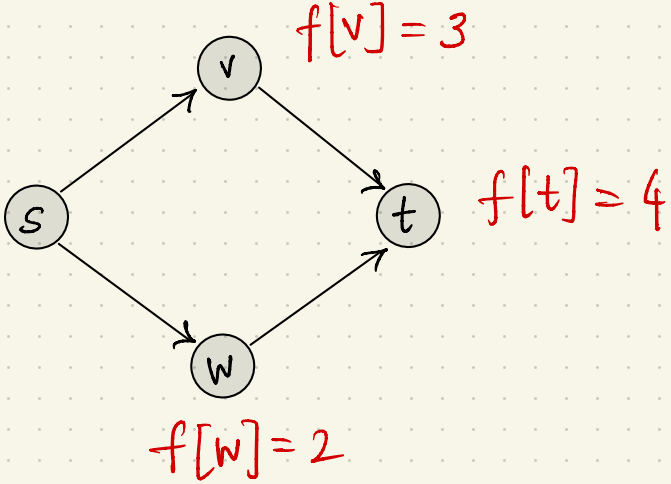


Call DFS on w

TOPOLOGICAL ORDERING via DFS

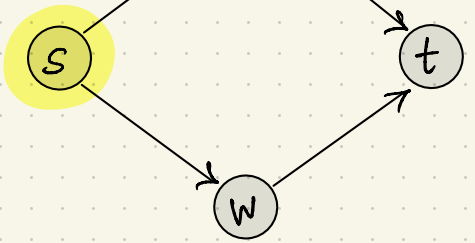


TOPOLOGICAL ORDERING via DFS



TOPOLOGICAL ORDERING via DFS

backtrack to s

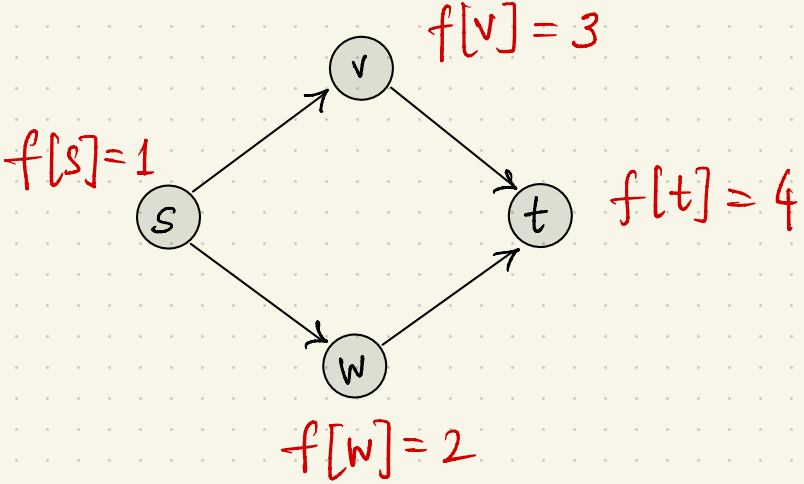


f[v] = 3

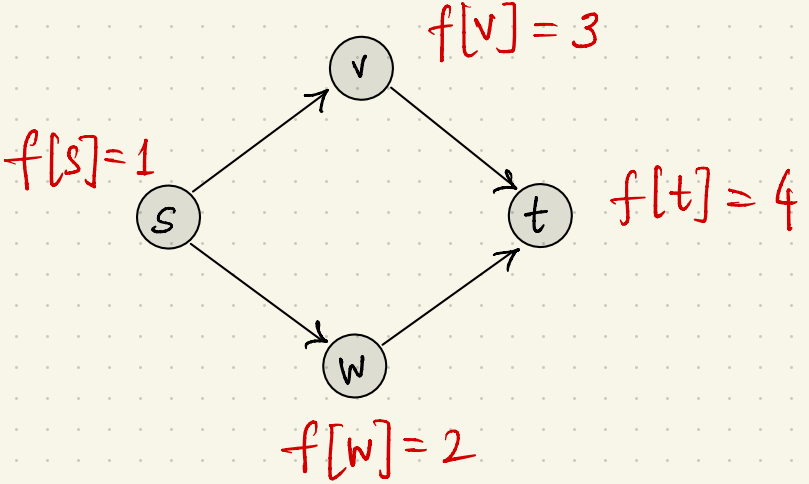
f[t] = 4

f[w] = 2

TOPOLOGICAL ORDERING via DFS



TOPOLOGICAL ORDERING via DFS



DFS - Loop runs out of vertices and stops!

TOPOLOGICAL ORDERING via DFS

Claim 1: DFS-Loop algorithm runs in $O(m+n)$ time.

TOPOLOGICAL ORDERING via DFS

Claim 1: DFS-Loop algorithm runs in $O(m+n)$ time.

Claim 2: Under DFS-Loop,

if $(u, v) \in E$ then $f[u] < f[v]$.

TOPOLOGICAL ORDERING via DFS

Claim 1: DFS-Loop algorithm runs in $O(m+n)$ time.

Claim 2: Under DFS-Loop,

if $(u, v) \in E$ then $f[u] < f[v]$.

Proof sketch: If u visited before $v \Rightarrow$

TOPOLOGICAL ORDERING via DFS

Claim 1: DFS-Loop algorithm runs in $O(m+n)$ time.

Claim 2: Under DFS-Loop,

if $(u, v) \in E$ then $f[u] < f[v]$.

Proof sketch: If u visited before v $\Rightarrow$ recursive call for v finishes before that of u

TOPOLOGICAL ORDERING via DFS

Claim 1: DFS-Loop algorithm runs in $O(m+n)$ time.

Claim 2: Under DFS-Loop,

if $(u, v) \in E$ then $f[u] < f[v]$.

Proof sketch: If u visited before v $\Rightarrow$ recursive call for v finishes before that of u

$\Rightarrow$ u's label $<$ v's label

TOPOLOGICAL ORDERING via DFS

Claim 1: DFS-Loop algorithm runs in $O(m+n)$ time.

Claim 2: Under DFS-Loop,

if $(u, v) \in E$ then $f[u] < f[v]$.

Proof sketch: If u visited before v $\Rightarrow$ recursive call for v finishes before that of u

$\Rightarrow$ u's label $<$ v's label

If v visited before u $\Rightarrow$

TOPOLOGICAL ORDERING via DFS

Claim 1: DFS-Loop algorithm runs in $O(m+n)$ time.

Claim 2: Under DFS-Loop,

if $(u,v) \in E$ then $f[u] < f[v]$.

Proof sketch: If u visited before v $\Rightarrow$ recursive call for v finishes before that of u

$\Rightarrow$ u's label $<$ v's label

If v visited before u $\Rightarrow$ no $v \rightsquigarrow u$ path (no cycle!)

TOPOLOGICAL ORDERING via DFS

Claim 1: DFS-Loop algorithm runs in $O(m+n)$ time.

Claim 2: Under DFS-Loop,

if $(u,v) \in E$ then $f[u] < f[v]$.

Proof sketch: If u visited before v $\Rightarrow$ recursive call for v finishes before that of u

$\Rightarrow$ u's label $<$ v's label

If v visited before u $\Rightarrow$ no $v \rightsquigarrow u$ path (no cycle!)

$\Rightarrow$ DFS(v) won't discover u

TOPOLOGICAL ORDERING via DFS

Claim 1: DFS-Loop algorithm runs in $O(m+n)$ time.

Claim 2: Under DFS-Loop,

if $(u,v) \in E$ then $f[u] < f[v]$.

Proof sketch: If u visited before v $\Rightarrow$ recursive call for v finishes before that of u

$\Rightarrow$ u's label $<$ v's label

If v visited before u $\Rightarrow$ no $v \rightsquigarrow u$ path (no cycle!)

$\Rightarrow$ DFS(v) won't discover u

$\Rightarrow$ v's recursive call finishes before u's. ▨