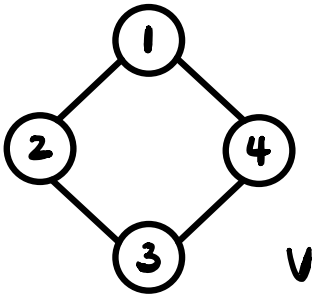
Ch.3 Graph Algorithms

3.1 Definitions

Def. $G = (V, E)$ — V: nodes, E: edges



$V = \{1, 2, 3, 4\}$

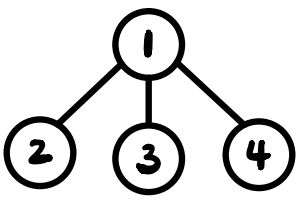
$E = \{(1,2), (1,4), (2,3), (3,4)\}$

undirected graph

$n = |V|$ # nodes

$m = |E|$ # edges

degree(v) = # edges associated w/ node v



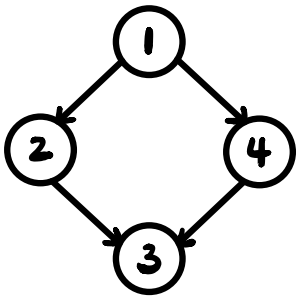
deg(1) = 3

deg(5) = 0

directed graph

indegree(v) = # edges pointing to v

outdegree(v) = # edges from v



indegree(1) = 0

outdegree(1) = 2

$V = \{1, 2, 3, 4\}$

$E = \{(1,2), (1,4), (2,3), (4,3)\}$

Data structures for storing graphs

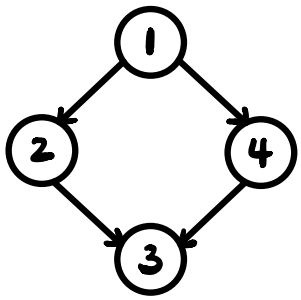
- adjacency matrix

$A =$

	1	2	3	4
1	0	1	0	1
2	0	0	1	0
3	0	0	0	0
4	0	0	1	0

$A[u,v] = 1$ if $(u,v) \in E$

- directed graph
- asymmetric



+ : $O(1)$ time to check $A[u,v]$

− : n^2 space

- in practice, $m \ll n^2$

− : $O(n)$ time to list all edges associated w/ a node

- usually $deg(v) < n$

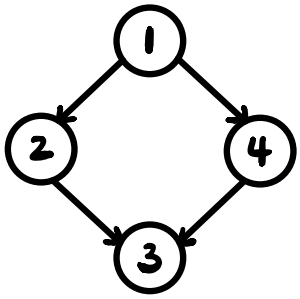
- adjacency list

node 1 head → 2 → 4

node 2 head → 3

node 3 head → null

node 4 head → 3



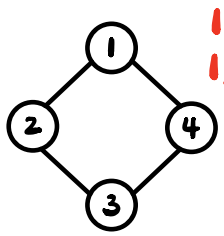
+ : $O(m+n)$ space

+ : $O(deg(v))$ time to enumerate edges associated w/ node v

− : $O(n^2)$ time to check $A[u,v]$

Paths & connectivity

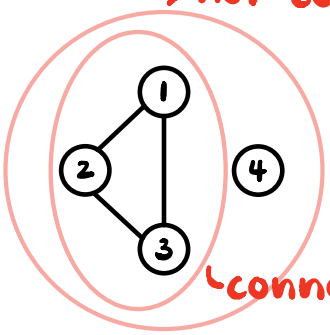
- **path**: sequence of nodes $v_1, v_2, \dots v_k$
  - $(v_i, v_{i+1}) \in E$
  - no repeated nodes (simple)



1, 2, 3
1, 2, 3, 4
⋮

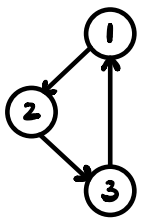
- nodes u and v are **connected**
  $\Leftrightarrow$ $\exists$ path between u and v

- **connected graph**: all nodes are connected to each other



- not connected
- connected

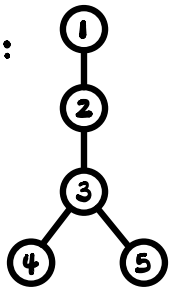
- **cycle**: sequence of nodes $v_1, v_2, \dots v_k = v_1$
  - $k > 2$
  - no repeated edges
  - no repeated nodes except $v_1 = v_k$

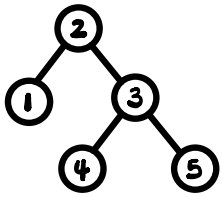


1, 2, 3, 1

Trees

- an undirected graph is a **tree** if:
  - connected
  - no cycles
- "root" the tree:





...

2 is an **ancestor** of 5
5 is a **descendant** of 2

- **unrooted tree**: leaf node w/ degree 1
- a tree has only 1 path between any two nodes

Pf. (by $\forall$) If $\exists$ 2 paths between u, v:

$$u \nearrow u_1 \to u_2 \to \dots \to u_k = v$$
$$u \searrow v_1 \to v_2 \to \dots \to v_\ell = v$$

To find $\bar{u}$: find the first i s.t. $u_i = v_i$ and $u_{i+1} \neq v_{i+1}$

For $\tilde{u}$: find the smallest $j, k > i$ s.t. $u_j = v_k = \tilde{u}_i$

$$\bar{u} = \tilde{u}_i \to u_{i+1} \to \dots \to u_j \to v_{k-1} \to v_{k-2} \to \dots \to v_i$$

$v_{k-1} = v_k$ (2 leaf nodes) $\Rightarrow$ cycle

$\to$ contradicts def. of tree

- each tree will have ≥ 2 nodes w/ deg. 1

Pf. Start from u, construct a path until cannot go anymore.

$$u \to u_1 \to \dots \to u_k$$
$$u_k \to v_1 \to \dots \to u_\ell$$

- every n-node tree has n-1 edges

3.2 Graph traversal and connectivity

- s-t connectivity : whether nodes s, t are connected

BFS (Breadth-First Search)

Level i : neighbors of level i-1 nodes that have not been visited

BFS:

$L_0 = \{s\}$, $T = \emptyset$

visited[i] = false $\forall$ i, visited[s] = true

i = 1

while $(L_{i-1} \neq \emptyset)$:

  $L_i = \emptyset$

  for $u \in L_{i-1}$:

    (*) for all v s.t. $(u, v) \in E$:

      if visited[v] = false:

        visited[v] = true

        add v to L_i

        add (u, v) to T

  i ± 1

BFS (queue implementation):

Q.push(s)

while (Q is not empty)

  u = Q.pop()

  for all v s.t. $(u, v) \in E$

    if visited[v] = false:

      visited[v] = true

      Q.push(v)

Properties

- L_i : nodes at distance i from s
- non-tree edges are either within a layer or between 2 adjacent layers
  - if (u,v) is a non-tree edge and $u \in L_i$, $v \in L_j$, then $|i-j| \leq 1$

use adj. list

(*) in $O(\deg(n))$

BFS time: $\sum \deg(n) = |E|$

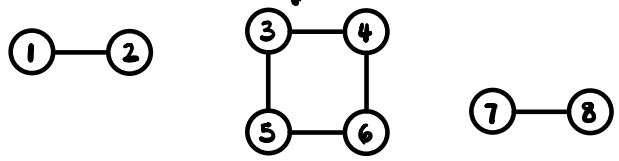
$O(|E|)$, $O(m)$

$O(n+m)$

use adj. matrix

Applications

- connected components



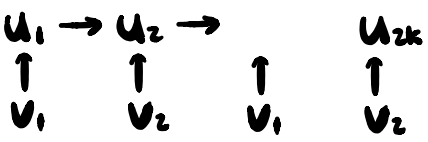
BFS(1): (1, 2)
BFS(3): (3, 4, 5, 6)
BFS(7): (7, 8)

- testing bipartiteness

Def. $G = (E, V)$ is bipartite if $G = V_1 \cup V_2$, $V_1 \cap V_2 = \phi$, all the edges are between V_1 and V_2

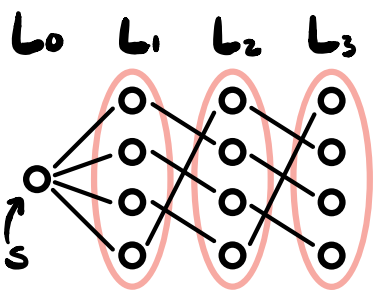
Lemma A bipartite graph has no odd cycle.

Pf.



- alg: run BFS, then check whether exists any edge within same level
  - yes ⇒ not bipartite
  - no ⇒ bipartite

Pf. if no edge within same level



...

$V_1 = L_0 \cup L_2 \cup L_4 \cdots$

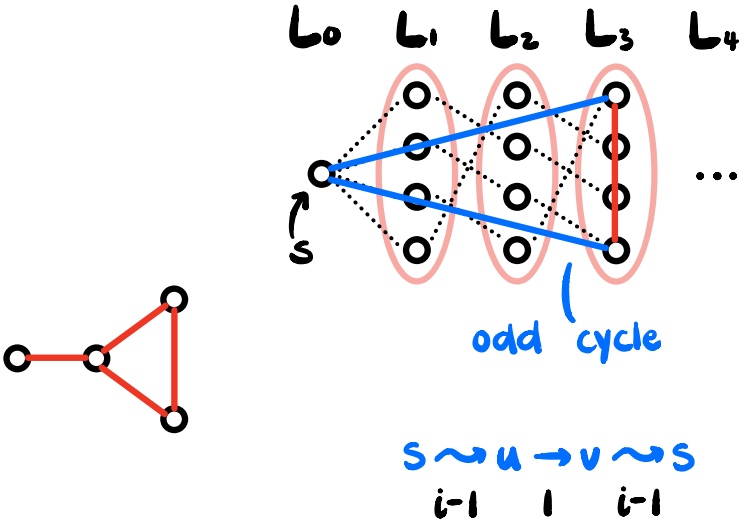
$V_2 = L_1 \cup L_3 \cup L_5 \cdots$

no edge within V_1 / V_2

BFS → no edges between L_i, L_j, $|i-j| \leq 1$

If ∃ edge within L_i:

show ∃ odd cycle



$s \rightsquigarrow u \to v \rightsquigarrow s$

$i-1$ 1 $i-1$

∃ a path $s \to u_1 \to u_2 \to \dots \to u_k = u$

a path $s \to v_1 \to v_2 \to \dots \to v_k = v$

Assume i is the last index s.t. $u_i = v_i$.

$u_i \to u_{i+1} \to \dots \to u_k \to v_k \to \dots \to v_i$

↑ odd cycle

DFS (Depth-First Search):

DFS(u):

```
visited[u] = true
for each v s.t. (u, v) ∈ E:
    if visited[v] = false :
        DFS(v)
```

Discover ordering

finishing: time when DFS(u) finished for u

Properties

- descendents of node u are all the nodes connected to u that are discovered after u
- if (u, v) is a nontree edge in DFS: either u is an ancestor of v or v is an ancestor of u

  Pf. assume u is discovered before v.

  v is connected to u

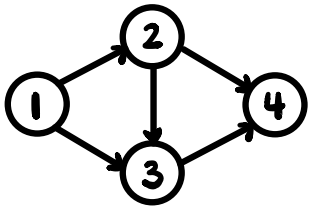
  $\Rightarrow$ v is a descendant of u

DFS (stack implementation):

```
A.push(s)
visited[i] = false ∀ i
while A is not empty :
    u ← A.pop()
    if visited[i] = false :
        visited[i] = true
        for v s.t. (u, v) ∈ E :
            A.push(v)
```

- articulation point

3.5 Directed Graph



$1 \rightsquigarrow 4$: a path from 1 to 4

$4 \not\rightsquigarrow 1$

Def. strongly connected: $u \rightsquigarrow v$ $\forall$ u, v in the graph

check a graph is SC:

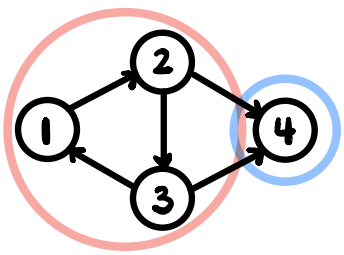
1. Pick any node s
2. Run BFS / DFS from s on G : R
   (nodes reachable from s)
3. Run BFS / DFS from s on G^{rev} : U ← G with each edge reversed
   (nodes that can reach s)

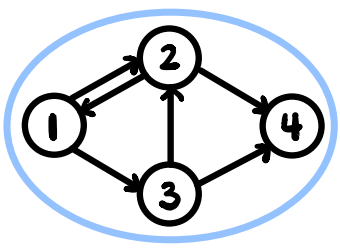
If $R \neq U$ or

strongly connected component (SCC)

Def. maximal strongly connected subgraph

↓ adding any other node will destroy SC of the subgraph





$V = V_1 \cup V_2 \cup \dots \cup V_k$ decompose graph to SCC

$V_i \cap V_j = \emptyset$

$\forall\ u, v \in V_i,\quad u \rightsquigarrow v$

decompose graph to SCCs:

while $\exists$ u not in any SCC:

$R \leftarrow$ DFS(u)

$U \leftarrow$ reverse DFS(u)

form a new SCC $R \cap U$

$\Theta(n(n+m))$ worst case

better algorithm: $O(n)$

2 DFS

Def. **Directed Acyclic Graph (DAG)**: directed graph w/o cycle.

multiple topological orderings

Def. **topological ordering**: an ordering of nodes $v_1, v_2, \dots v_n$ s.t. all edges are pointing forward

$\forall\ (v_i, v_j) \in E,\ i<j$

Thm. A directed graph G has a topological ordering $\Rightarrow$ G is a DAG.

Pf. (by γ)

Assume G has T-order.

If G has cycle $u_1 u_2 \dots u_k u_1$,

let u_i be the last one in T-order.

$\Rightarrow$ $u_i \to u_{i+1}$ will be a backward edge. γ

Lemma G is a DAG $\Rightarrow$ $\exists$ a node in G with no incoming edge.

Pf. Can find a cycle in G if all nodes have indegree ≥ 1

Start from an arbitrary node u; keep finding nodes "backward".

$u_0 \leftarrow u_1 \leftarrow u_2 \leftarrow u_3 \leftarrow \dots$

Can keep going until visiting a node a second time

$\Rightarrow$ find a cycle in G.

Thm. G is a DAG $\Rightarrow$ G has a topological ordering.

Pf. (by induction)

Base case: n=1: 1 node ✓

IH: Suppose every DAG with n-1 nodes has a topological ordering.

IS: For DAG w/ n nodes:

$\exists$ node u w/ indegree = 0.

Remove u; G - {u} = DAG with n-1 nodes.

G - {u} has a topological ordering $v_1 \dots v_n$.

Then $u, v_1, \dots v_n$ is a valid topological ordering for G.

Topological sort:

for $i = 1, 2, \ldots n$:

$u \leftarrow$ node w/ indegree = 0

make u the next node in T-order

remove u from the graph

↳ remove all associated edges: $O(n+m)$

array indegree[i] $i = 1, 2, \ldots n$

initial indegree array $O(n+m)$

push i to L if indegree[i] = 0

for $i = 1, 2, \ldots n$:

$u \leftarrow$ L.pop()

append u to T-order

for v s.t. $(u, v) \in E$

indegree[v] --

if indegree[v] = 0:

L.push(v)

check if G is a DAG:

if can't find any node w/ indegree = 0 in the loop:

G is not a DAG: