CSE 431/531: Algorithm Analysis and Design (Spring 2019)
Graph Basics

Lecturer: Shi Li

Department of Computer Science and Engineering
University at Buffalo

Outline

Examples of Graphs

Figure: Road Networks

Figure: Internet

Figure: Social Networks

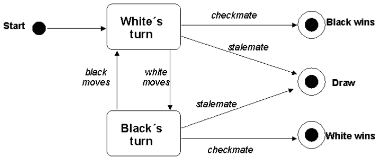
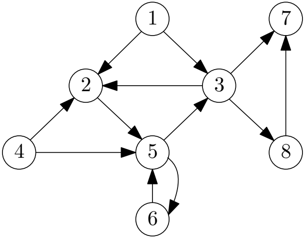


Figure: Transition Graphs

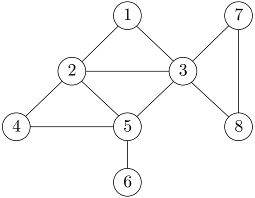
(Undirected) Graph $G = (V, E)$



- V: set of vertices (nodes);
  - $V = \{1, 2, 3, 4, 5, 6, 7, 8\}$
- E: pairwise relationships among V;
  - (undirected) graphs: relationship is symmetric, E contains subsets of size 2
  - $E = \{\{1,2\}, \{1,3\}, \{2,3\}, \{2,4\}, \{2,5\}, \{3,5\}, \{3,7\}, \{3,8\}, \{4,5\}, \{5,6\}, \{7,8\}\}$

Abuse of Notations

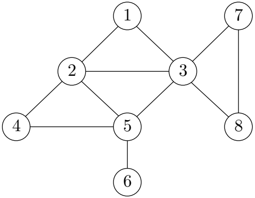
- For (undirected) graphs, we often use (i,j) to denote the set $\{i,j\}$.
- We call (i,j) an unordered pair; in this case $(i,j) = (j,i)$.



- $E = \{(1,2),(1,3),(2,3),(2,4),(2,5),(3,5),(3,7),(3,8),$
$(4,5),(5,6),(7,8)\}$

- Social Network : Undirected
- Transition Graph : Directed
- Road Network : Directed or Undirected
- Internet : Directed or Undirected

Representation of Graphs



1: 2 → 3

2: 1 → 3 → 4 → 5

3: 1 → 2 → 5 → 7 → 8

4: 2 → 5

5: 2 → 3 → 4 → 6

6: 5

7: 3 → 8

8: 3 → 7

- Adjacency matrix
  - $n \times n$ matrix, $A[u, v] = 1$ if $(u, v) \in E$ and $A[u, v] = 0$ otherwise
  - A is symmetric if graph is undirected
- Linked lists
  - For every vertex v, there is a linked list containing all neighbours of v.

Comparison of Two Representations

- Assuming we are dealing with undirected graphs
- n: number of vertices
- m: number of edges, assuming $n-1 \leq m \leq n(n-1)/2$
- d_v: number of neighbors of v

	Matrix	Linked Lists
memory usage	$O(n^2)$	$O(m)$
time to check $(u,v) \in E$	$O(1)$	$O(d_u)$
time to list all neighbours of v	$O(n)$	$O(d_v)$

Outline

Connectivity Problem

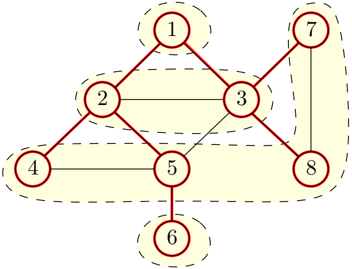
Input: graph $G = (V, E)$, (using linked lists)
two vertices $s, t \in V$

Output: whether there is a path connecting s to t in G

- Algorithm: starting from s, search for all vertices that are reachable from s and check if the set contains t
  - Breadth-First Search (BFS)
  - Depth-First Search (DFS)

Breadth-First Search (BFS)

- Build layers $L_0, L_1, L_2, L_3, \cdots$
- $L_0 = \{s\}$
- L_{j+1} contains all nodes that are not in $L_0 \cup L_1 \cup \cdots \cup L_j$ and have an edge to a vertex in L_j



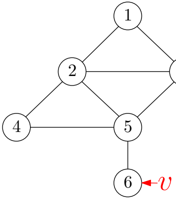
Implementing BFS using a Queue

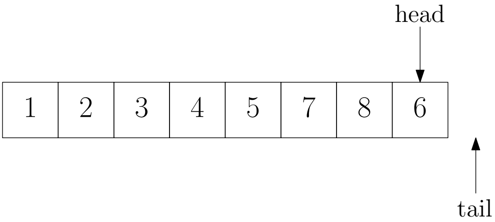
BFS(s)

```
head ← 1, tail ← 1, queue[1] ← s
mark s as "visited" and all other vertices as "unvisited"
while head ≥ tail
    v ← queue[tail], tail ← tail + 1
    for all neighbours u of v
        if u is "unvisited" then
            head ← head + 1, queue[head] = u
            mark u as "visited"
```

- Running time: $O(n+m)$.

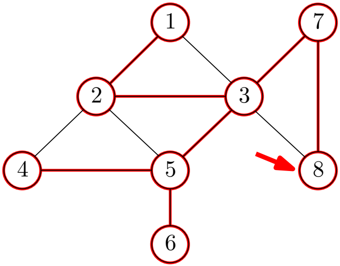
Example of BFS via Queue





Depth-First Search (DFS)

- Starting from s
- Travel through the first edge leading out of the current vertex
- When reach an already-visited vertex ("dead-end"), go back
- Travel through the next edge
- If tried all edges leading out of the current vertex, go back



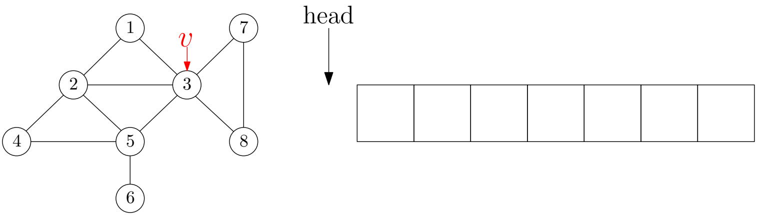
Implementing DFS using a Stack

DFS(s)

$head \leftarrow 1, stack[1] \leftarrow s$
mark all vertices as "unexplored"
while head ≥ 1
    $v \leftarrow stack[head], head \leftarrow head - 1$
    if v is unexplored then
        mark v as "explored"
        for all neighbours u of v
            if u is not explored then
                $head \leftarrow head + 1, stack[head] = u$

- Running time: $O(n + m)$.

Example of DFS using Stack



explored vertices: 1 2 3 5 4 6 7 8

Implementing DFS using Recurrsion

DFS(s)

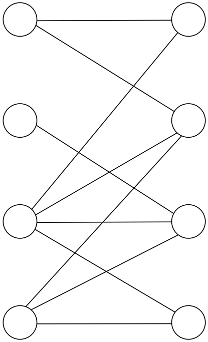
1. mark all vertices as "unexplored"
2. recursive-DFS(s)

recursive-DFS(v)

1. if v is explored then return
2. mark v as "explored"
3. for all neighbours u of v
4.   recursive-DFS(u)

Outline

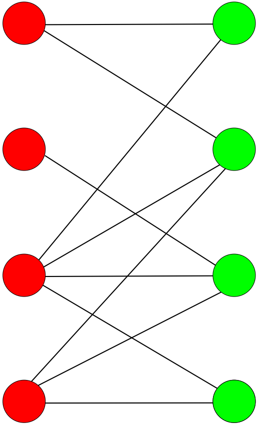
Def. A graph $G = (V, E)$ is a bipartite graph if there is a partition of V into two sets L and R such that for every edge $(u, v) \in E$, we have either $u \in L, v \in R$ or $v \in L, u \in R$.

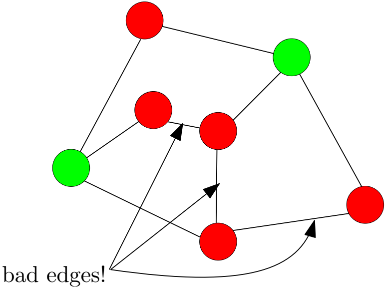


Testing Bipartiteness

- Taking an arbitrary vertex $s \in V$
- Assuming $s \in L$ w.l.o.g
- Neighbors of s must be in R
- Neighbors of neighbors of s must be in L
- $\cdots$
- Report "not a bipartite graph" if contradiction was found
- If G contains multiple connected components, repeat above algorithm for each component

Test Bipartiteness





Testing Bipartiteness using BFS

BFS(s)

$head \leftarrow 1, tail \leftarrow 1, queue[1] \leftarrow s$

mark s as "visited" and all other vertices as "unvisited"

$color[s] \leftarrow 0$

while head $\geq$ tail

    $v \leftarrow queue[tail], tail \leftarrow tail + 1$

    for all neighbours u of v

        if u is "unvisited" then

            $head \leftarrow head + 1, queue[head] = u$

            mark u as "visited"

            $color[u] \leftarrow 1 - color[v]$

        elseif $color[u] = color[v]$ then

            print("G is not bipartite") and exit

Testing Bipartiteness using BFS

1. mark all vertices as "unvisited"
2. for each vertex $v \in V$
3.   if v is "unvisited" then
4.   test-bipartiteness(v)
5. print("G is bipartite")

Obs. Running time of algorithm $= O(n + m)$

Homework problem: using DFS to implement test-bipartiteness.

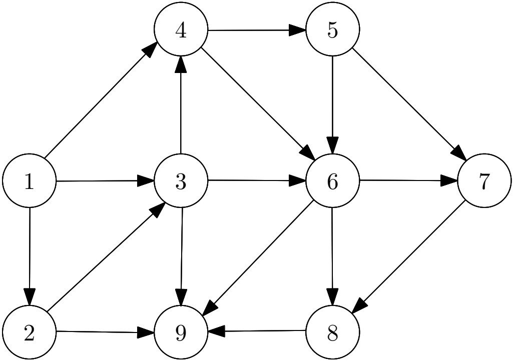
Outline

Topological Ordering Problem

Input: a directed acyclic graph (DAG) $G = (V, E)$

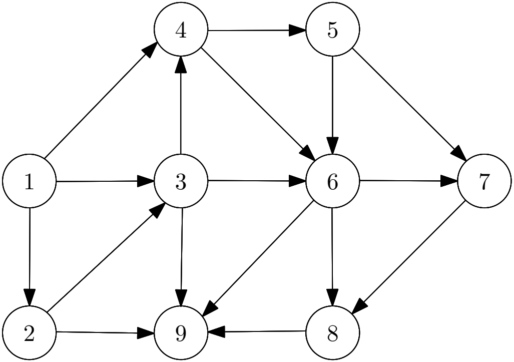
Output: 1-to-1 function $\pi : V \to \{1, 2, 3 \cdots, n\}$, so that

- if $(u, v) \in E$ then $\pi(u) < \pi(v)$



Topological Ordering

- Algorithm: each time take a vertex without incoming edges, then remove the vertex and all its outgoing edges.



Topological Ordering

- Algorithm: each time take a vertex without incoming edges, then remove the vertex and all its outgoing edges.

Q: How to make the algorithm as efficient as possible?

A:

- Use linked-lists of outgoing edges
- Maintain the in-degree d_v of vertices
- Maintain a queue (or stack) of vertices v with $d_v = 0$

topological-sort(G)

let $d_v \leftarrow 0$ for every $v \in V$
for every $v \in V$
    for every u such that $(v, u) \in E$
        $d_u \leftarrow d_u + 1$
$S \leftarrow \{v : d_v = 0\}, i \leftarrow 0$
while $S \neq \emptyset$
    $v \leftarrow$ arbitrary vertex in S, $S \leftarrow S \setminus \{v\}$
    $i \leftarrow i + 1$, $\pi(v) \leftarrow i$
    for every u such that $(v, u) \in E$
        $d_u \leftarrow d_u - 1$
        if $d_u = 0$ then add u to S
if $i < n$ then output "not a DAG"

- S can be represented using a queue or a stack
- Running time $= O(n + m)$