CSE 431/531: Algorithm Analysis and Design (Spring 2020)

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

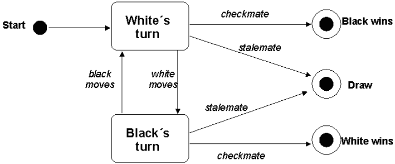
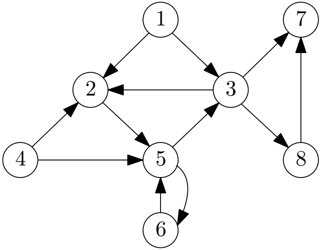


Figure: Transition Graphs

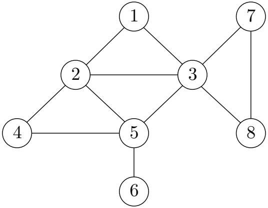
(Undirected) Graph $G = (V, E)$



- 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

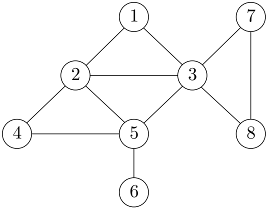
- For (undirected) graphs, we often use (i, j) to denote the set $\{i, j\}$.
- We call (i, j) an unordered pair; in this case $(i, j) = (j, i)$.

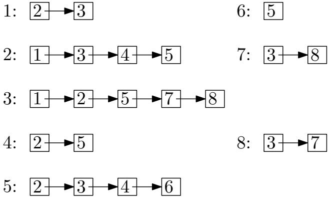


- $E = \{(1,2),(1,3),(2,3),(2,4),(2,5),(3,5),(3,7),(3,8),$ $(4,5),(5,6),(7,8)\}$

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

Representation of Graphs





- 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 $n - 1 \le m \le n(n-1)/2$
- 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

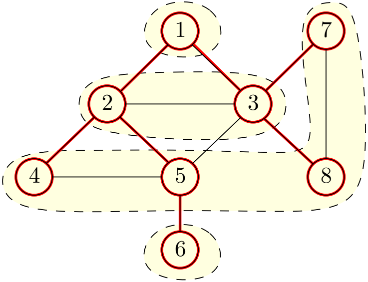
Input: graph $G = (V, E)$, (using linked lists)
two vertices $s, t \in V$

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



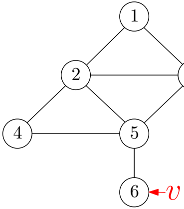
Implementing BFS using a Queue

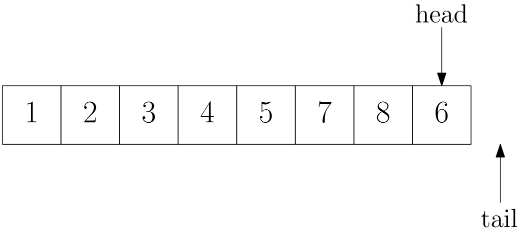
BFS(s)

1. $head \leftarrow 1, tail \leftarrow 1, queue[1] \leftarrow s$
2. mark s as "visited" and all other vertices as "unvisited"
3. while head $\geq$ tail
4. $v \leftarrow queue[tail], tail \leftarrow tail + 1$
5.   for all neighbours u of v
6.   if u is "unvisited" then
7. $head \leftarrow head + 1, queue[head] = u$
8.   mark u as "visited"

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

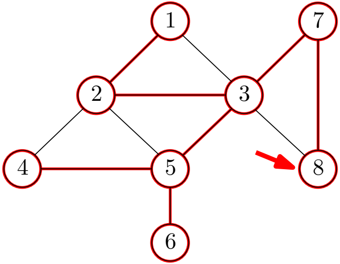
Example of BFS via Queue





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 Recurrsion

DFS(s)

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

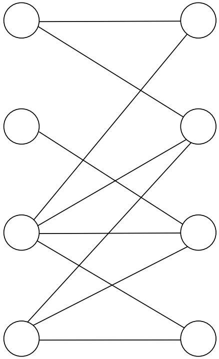
recursive-DFS(v)

1. mark v as "visited"
2. for all neighbours u of v
3.   **if** u is unvisited **then** recursive-DFS(u)

Outline

Testing Bipartiteness: Applications of BFS

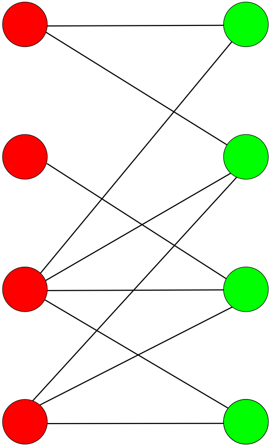
Def. A graph $G = (V, E)$ is a bipartite graph if there is a partition of V into two sets L and R such that for every edge $(u, v) \in E$, we have either $u \in L, v \in R$ or $v \in L, u \in R$.

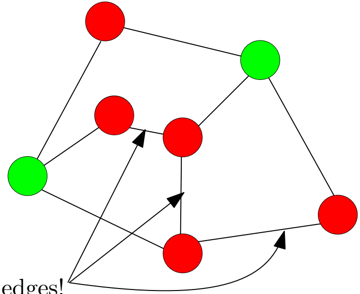


Testing Bipartiteness

- 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 $= O(n+m)$

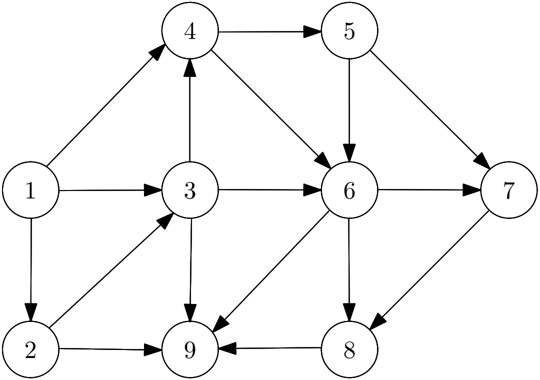
Outline

Topological Ordering Problem

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

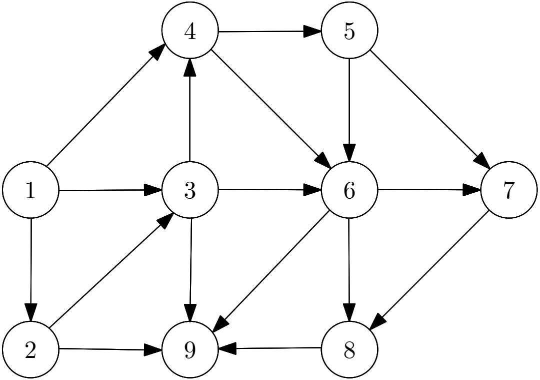
Output: 1-to-1 function $\pi : V \to \{1, 2, 3 \cdots, n\}$, so that

- 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 $= O(n + m)$

S as a Queue or a Stack

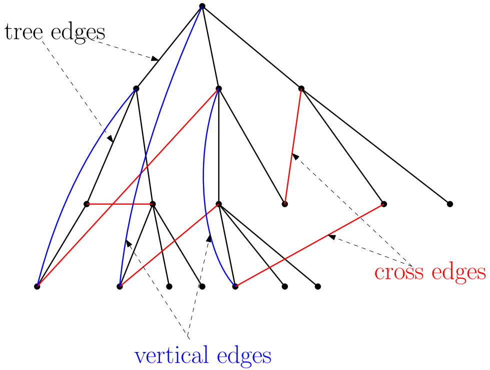
DS	Queue	Stack
Initialization	$head \leftarrow 0$, $tail \leftarrow 1$	$top \leftarrow 0$
Non-Empty?	$head \geq tail$	$top > 0$
Add(v)	$head \leftarrow head + 1$ $S[head] \leftarrow v$	$top \leftarrow top + 1$ $S[top] \leftarrow v$
Retrieve v	$v \leftarrow S[tail]$ $tail \leftarrow tail + 1$	$v \leftarrow S[top]$ $top \leftarrow top - 1$

Outline

Type of edges with respect to a tree

Given a graph $G = (V, E)$ and a rooted tree T in G, edges in G can be one of the three types:

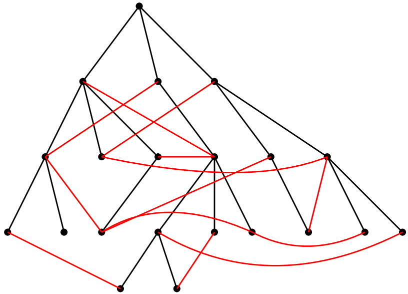
- Tree edges: edges in T
- Cross edges (u, v): u and v do not have an ancestor-descendant relation
- Vertical edges (u, v): u is an ancestor of v, or v is an ancestor of u



Properties of a BFS Tree

Given a tree BFS tree T of a graph G,

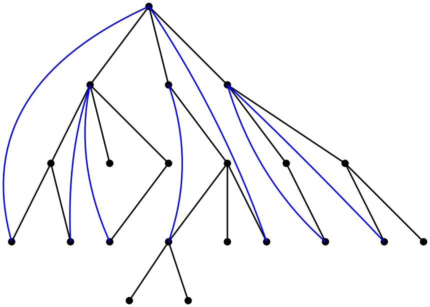
- Can there be vertical edges?
- No.
- Can there be cross edges (u, v) with u and v 2 levels apart?
- No.
- For any cross edge (u, v), u and v are at most 1 level apart.



Properties of a DFS Tree

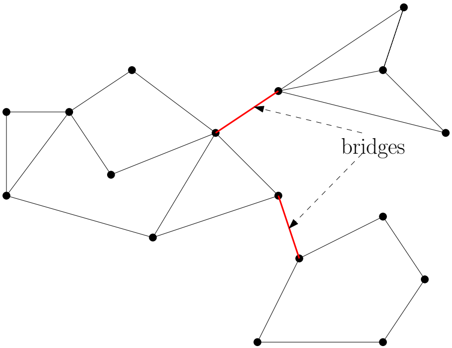
Given a tree DFS tree T of a graph G,

- Can there be cross edges?
- No.
- All non-tree edges are vertical edges.

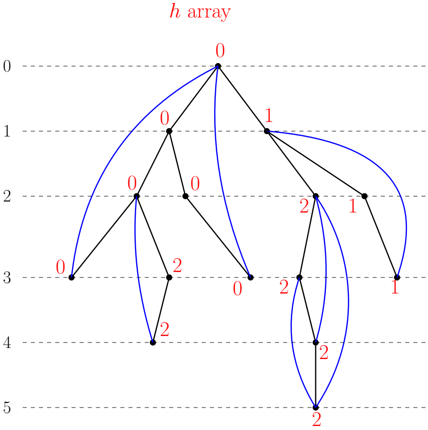


Bridges in a Graph

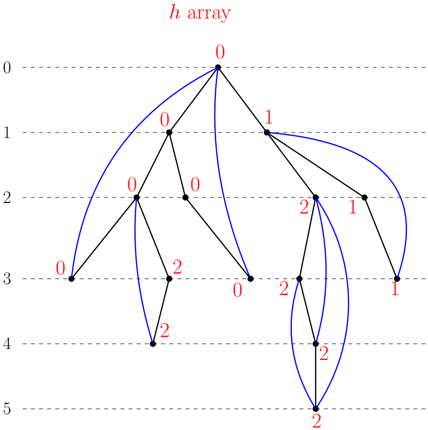
Def. Given a connected graph $G = (V, E)$, an edge $e \in E$ is called a bridge if the graph $G = (V, E \setminus \{e\})$ is disconnected.



- There are only tree edges and vertical edges
- Vertical edges are not bridges
- A tree edge (v, u) is not a bridge if some vertical edge jumping from below u to above v
- Other tree edges are bridges



- $level(v)$: the level of vertex v in DFS tree
- T_v: the sub tree rooted at v
- $h(v)$: the smallest level that can be reached using a vertical edge from vertices in T_v
- $(parent(u), u)$ is a bridge if $h(u) \geq level(u)$.



recursive-DFS(v)

```
mark v as "visited"
h(v) ← ∞
for all neighbours u of v
    if u is unvisited then
        level(u) ← level(v) + 1
        recursive-DFS(u)
        if h(u) ≥ level(u) then claim (v, u) is a bridge
        if h(u) < h(v) then h(v) ← h(u)
    else if level(u) < level(v) − 1 then
        if level(u) < h(v) then h(v) ← level(u)
```

Finding_Bridges

1. mark all vertices as "unvisited"
2. **for** every $v \in V$ **do**
3.   **if** v is unvisited **then**
4. $level(v) \leftarrow 0$
5.   recursive-DFS(v)