CS 341: ALGORITHMS

Lecture 13: graph algorithms II – finishing BFS, depth first search
Readings: see website

Trevor Brown
https://student.cs.uwaterloo.ca/~cs341
trevor.brown@uwaterloo.ca
BFS APPLICATION: TESTING WHETHER A GRAPH IS BIPARTITE
A graph is **bipartite** if the nodes can be **partitioned** into sets $R$ and $B$ such that **each edge** has one endpoint in $R$ and one endpoint in $B$. 

**Diagram:**

- Graph $G$ with nodes 1, 2, 3, 4, 5, 6, 7, 8.
- Nodes 1, 2, 3, 4 are in set $R$.
- Nodes 5, 6, 7, 8 are in set $B$.
- Each edge connects a node in $R$ to a node in $B$. 

**Directed Graph:***

- Directed graph formed by removing the bidirectional edges and replacing them with directed edges from the set $R$ to the set $B$. 
- Nodes 1, 2, 3, 4 in set $R$ are connected to nodes 5, 6, 7, 8 in set $B$.
CRUCIAL PROPERTY: NO ODD CYCLES

- **Claim:** A graph is bipartite if and only if it does not contain an odd length cycle.

What happens if I create an odd length cycle?
PROOF
PART 1: ODD CYCLE ⇒ NOT BIPARTITE

- Suppose there is an odd length cycle \( v_1, v_2, ..., v_{2k+1}, v_1 \)

And so on, alternating...

And \( v_4 \in B \)

And \( v_3 \in R \)

And \( v_2 \in B \)

Then we must have \( v_2 \in B \)

(Or there will be an edge \((v_1, v_2)\) with two endpoints in \( R \))

WLOG let \( v_1 \in R \)

And finally \( v_{2k+1} \in R \)!!

Both endpoints in \( R \)!

Contradiction!
PROOF

PART 2: ALL CYCLES HAVE EVEN LENGTH ⇒ BIPARTITE

- Let \( v_i \) be any node, and \( d(v) \) be the distance from \( v_i \) to \( v \)
- Partition nodes by even vs odd distances

\[ G \]

\[ R = \text{odd } d(v) \quad B = \text{even } d(v) \]

WTP: no edge between red nodes
no edge between blue nodes
BAD EDGES MEAN ODD CYCLES

- **Claim:** if there were an edge between red nodes, or between blue nodes, there would be an **odd length cycle**
- WLOG suppose for contradiction \( u, v \in E \) where \( u, v \in R \)
- Since \( u, v \in R \), distances \( d(u) \) and \( d(v) \) from \( v_i \) are **both odd**

Recall \( d(u) = \) length of shortest path \( v_i \to \cdots \to u \)

\[
\begin{align*}
\cdots & \quad d(u) = \text{odd} \\
\vdots & \\
\cdots & \quad \cdots \\
\end{align*}
\]

The combined path \( v_i \to \cdots \to u \to v \to \cdots \to v_i \) forms a cycle

And its length is \( d(u) + 1 + d(v) \) which is odd!

\[
\begin{align*}
\cdots & \quad d(v) = \text{odd} \\
\vdots & \\
\cdots & \quad \cdots \\
\end{align*}
\]

...and \( d(v) \) the shortest path \( v_i \to \cdots \to v \)

So there is no edge \( (u, v) \) where \( u, v \in R \) (case \( B \) is similar)
ALGORITHM FOR TESTING BIPARTITENESS

Bipartition(adj[1..n])
  colour[1..n] = [white, ..., white]
  dist[1..n] = [infty, ..., infty]
  for start = 1..n
    if colour[start] is white
      BFS(adj, start, colour, dist)
  for edge in adj
    let u and v be endpoints of edge
    if (dist[u]%2) == (dist[v]%2) then
      return NotBipartite
  B = nodes u with even dist[u]
  R = nodes u with odd dist[u]
  return B, R

Call BFS on each component to calculate distances for each node
Modified BFS that reuses the same colour array and same dist array
If both even or both odd
Return an actual bipartition
Runtime complexity?
Can be done in $O(n + m)$
Bread first search

DEPTH FIRST SEARCH

Depp first search
DEPTH-FIRST SEARCH OF A DIRECTED GRAPH

A depth-first search uses a stack (or recursion) instead of a queue.
We define predecessors and colour vertices as in BFS.
It is also useful to specify a discovery time \(d[v]\) and a finishing time \(f[v]\)
for every vertex \(v\).
We increment a time counter every time a value \(d[v]\) or \(f[v]\) is assigned.
We eventually visit all the vertices, and the algorithm constructs a
depth-first forest.
**DEPTH FIRST SEARCH ALGORITHM**

```plaintext
1 global variables:
2    pred[1..n] = [null, null, ..., null]
3    colour[1..n] = [white, white, ..., white]
4    d[1..n] = [0, 0, ..., 0] // discovery times
5    f[1..n] = [0, 0, ..., 0] // finish times
6    time = 0

7 DepthFirstSearch(adj[1..n])
8     for v = 1..n
9        if colour[v] == white
10           DFSVisit(v)
11
12 DFSVisit(adj[1..n], v)
13     colour[v] = gray
14     time = time + 1
15     d[v] = time
16
17     for each w in adj[v]
18        if colour[w] == white
19           pred[w] = v
20           DFSVisit(w)
21
22     colour[v] = black
23     time = time + 1
24     f[v] = time
```

*Example execution starting at node 1*

- **DFSVisit(1)**
  - time = 0
  - d[1] = 1
  - f[1] = 10

- **DFSVisit(6)**
  - time = 12
  - d[6] = 11
  - f[6] = 12

Nodes: 1, 2, 3, 4, 5, 6

- **Node 1**: d[1] = 1, f[1] = 10

**Not White**

- Node 3
- Node 2
- Node 5
- Node 6

**Depth First Search**

- **Time Stamps**
  - time = 0
  - time = 1
  - time = 2
  - time = 3
  - time = 4
  - time = 5
  - time = 6
  - time = 7
  - time = 8
  - time = 9
  - time = 10
  - time = 11
  - time = 12

**Predicates**

- pred[1..n] = [null, null, ..., null]

**Colouring**

- colour[1..n] = [white, white, ..., white]

**Discovery Times**

- d[1..n] = [0, 0, ..., 0]

**Finish Times**

- f[1..n] = [0, 0, ..., 0]
DFS TREE / FOREST

- As in breadth first search, \texttt{pred[]} array induces a forest.
- Let's match the graph's edge directions (opposite from \texttt{pred}).

**Graph**

- **1**
  - \(d[1]=1\)
  - \(f[1]=10\)
- **2**
  - \(d[2]=2\)
  - \(f[2]=9\)
- **3**
  - \(d[3]=3\)
  - \(f[3]=4\)
- **4**
  - \(d[4]=5\)
  - \(f[4]=6\)
- **5**
  - \(d[5]=7\)
  - \(f[5]=8\)
- **6**
  - \(d[6]=11\)
  - \(f[6]=12\)

**DFS forest**

- **Tree 1**
  - \(1\)
  - \(3\)
  - \(5\)
  - \(6\)
- **Tree 2**
  - \(2\)
  - \(4\)
  - \(5\)

**DepthFirstSearch(adj[1..n])**

\begin{verbatim}
for v = 1..n
    if colour[v] == white
        DFSVisit(v)
\end{verbatim}

Each top level DFSVisit call is the root of a tree.

Recall: DFSVisit(1), DFSVisit(6)
BASIC DFS PROPERTIES TO REMEMBER

- Nodes start **white**
- A node \( v \) turns **gray** when it is **discovered**, which is when the first call to \( \text{DFSVisit}(v) \) happens
- **After** \( v \) is turned **gray**, we recurse on its neighbours
- **After** recursing on all neighbours, we turn \( v \) **black**
  - Recursive calls on neighbours end before \( \text{DFSVisit}(v) \) does, so the neighbours of \( v \) turn black before \( v \)

Also gets a **discovery time** \( d[v] \) at this point

Also gets a **finish time** \( f[v] \) at this point
RUNTIME COMPLEXITY OF DFS (FOR ADJ. LISTS)

Home exercise: complexity with adjacency matrix?

Only called on a white node, and immediately colours the node gray
So called once per node!

Each call iterates over the neighbours. Effectively: “for each node, for each neighbour, do O(1) work + recurse.”

Total O(n+m) iterations over all recursive calls. Total O(n+m) runtime!
CLASSIFYING EDGE IN DFS

- If \( \text{pred}[v] = u \), then: \((u, v)\) is a **tree edge**
- Else if \( v \) is a descendent of \( u \) in the DFS forest: **forward edge**
- Else if \( v \) is an ancestor of \( u \) in the DFS forest: **back edge**
- Else: \((u, v)\) is a **cross edge**

![Graph and DFS forest]

Can we classify edges **without** inspecting the DFS forest? Perhaps using \( d[\ldots] \), \( f[\ldots] \), \textcolor{red}{	extbf{colour}}[\ldots]?
**DEFINITIONS**

- **Definition:** we use $I_u$ to denote $(d[u], f[u])$, which we call the **interval of** $u$

- **Definition:** $v$ is **white-reachable from** $u$ if there is a path from $u$ to $v$ containing **only white nodes** (excluding $u$)
EXPLORING D[], F[] AND COLOUR[]

- **Observe:** every node \( v \) that is white-reachable from \( u \) when we first call \( DFSVisit(u) \) becomes **gray after \( u \) and black before \( u \)** (so \( I_v \) is **nested inside** \( I_u \))

Start \( DFSVisit(u) \),
- colour \( u \) grey, and
- set \( u \)’s discovery time

Perform \( DFSVisit \) calls recursively...

Colour \( u \) black,
- set \( u \)’s finish time
- and return from \( DFSVisit(u) \)

Consider the **tree of recursive calls**
- rooted at \( DFSVisit(u) \).

\( v \) is discovered by a call in this tree
- **iff** \( I_v \) is nested inside \( I_u \)

**iff** \( v \) is a descendent of \( u \)
- in the DFS forest

**iff** \( v \) turns grey after \( u \) and black before \( u \)

**iff** \( v \) is white-reachable from \( u \)
- when \( DFSVisit(u) \) is called
SUMMARIZING IN A THEOREM

• **Theorem:** Let \( u, v \) be any nodes. The following statements are all equivalent.
  - \( v \) is white-reachable from \( u \) when we call \( DFSVisit(u) \)
  - \( v \) turns grey after \( u \) and black before \( u \)
  - (discovery/finish time interval \( I_v \) is nested inside \( I_u \))
  - \( v \) is discovered during \( DFSVisit(u) \)
  - \( v \) is a descendant of \( u \) in the DFS forest)
# Classifying Edge Types in DFS

DFS inspects **every edge** in the graph. **When** DFS inspects an edge \( \{u, v\} \), the colour of \( v \) and relationship between the intervals of \( u \) and \( v \) determine the edge type.

<table>
<thead>
<tr>
<th>edge type</th>
<th>colour of ( v )</th>
<th>discovery/finish times</th>
</tr>
</thead>
<tbody>
<tr>
<td>tree</td>
<td>Q1?</td>
<td>Q2?</td>
</tr>
<tr>
<td>forward</td>
<td>Q4?</td>
<td>Q3?</td>
</tr>
<tr>
<td>back</td>
<td>Q6?</td>
<td>Q5?</td>
</tr>
<tr>
<td>cross</td>
<td>Q8?</td>
<td>Q7?</td>
</tr>
</tbody>
</table>

**Recall:** \((v \text{ is discovered during } \text{DFSVisit}(u))\)
- \((v \text{ is white-reachable from } u \text{ when we call } \text{DFSVisit}(u))\)
- \((v \text{ is a descendant of } u \text{ in the DFS forest})\)
- \((v \text{ turns grey after } u \text{ and black before } u)\)
- \((I_v \text{ nested inside } I_u)\)

\(v\) discovered **during** \(\text{DFSVisit}(u)\) **but not directly** from \( u \) (or \( \{u, v\} \) would be a tree edge)

So when \(\text{DFSVisit}(u)\) inspects \(\{u, v\}\), \(v\) **cannot** be white

\(v\) is a **child** of \(u\) in the DFS tree

\(v\) is a **descendent** of \(u\)

\(v\) is an **ancestor** of \(u\)

\(v\) is **not** a descendent, and **not** an ancestor

\(v\) **is already discovered**!

... by another recursive call that \(\text{DFSVisit}(u)\) makes when it inspects a previous edge

That call **terminates** before \(\text{DFSVisit}(u)\) inspects \(\{u, v\}\)

And it colors \(v\) **black**!
USEFUL FACT: PARENTHESIS THEOREM

- **Theorem:** for each pair of nodes $u, v$
  the intervals of $u$ and $v$ are either disjoint or nested.

- **Proof:** Suppose the intervals are not disjoint.
  - Then either $d[v] \in I_u$ or $d[u] \in I_v$
  - WLOG suppose $d[v] \in I_u$
  - Then $v$ is discovered during $DFSVisit(u)$
  - So, $v$ must turn gray after $u$ and black before $u$
  - So $f[v] < f[u]$
  - So the intervals are nested. QED
**CLASSIFYING EDGE TYPES IN DFS**

DFS inspects **every edge** in the graph. When DFS inspects an edge \( \{u, v\} \), the colour of \( v \) and relationship between the intervals of \( u \) and \( v \) determine the **edge type**.

### Edge Types

<table>
<thead>
<tr>
<th>edge type</th>
<th>colour of ( v )</th>
<th>discovery/finish times</th>
</tr>
</thead>
<tbody>
<tr>
<td>tree</td>
<td>white</td>
<td>( d[u] &lt; d[v] &lt; f[v] &lt; f[u] )</td>
</tr>
<tr>
<td>forward</td>
<td>black</td>
<td>( d[u] &lt; d[v] &lt; f[v] &lt; f[u] )</td>
</tr>
<tr>
<td>back</td>
<td>gray</td>
<td>( d[v] &lt; d[u] &lt; f[u] &lt; f[v] )</td>
</tr>
<tr>
<td>cross</td>
<td>Q8?</td>
<td>Q7?</td>
</tr>
</tbody>
</table>

**Recall:**  
\( v \) is discovered during \( DFSVisit(u) \)  
\( \Leftrightarrow \) \( v \) is **white-reachable** from \( u \) when we call \( DFSVisit(u) \)  
\( \Leftrightarrow \) \( v \) is a **descendant of** \( u \) in the DFS forest  
\( \Leftrightarrow \) \( v \) turns grey after \( u \) and black before \( u \)  
\( \Leftrightarrow \) \( I_v \) nested inside \( I_u \)

---

**So,** \( I_v \) **must be earlier.**

If \( I_u \) were earlier, then \( v \) would be **discovered before** \( u \) **finishes** (because of edge \( \{u, v\} \)), so intervals would not be disjoint!

**Intervals** \( I_u \) and \( I_v \) **must be disjoint.** But which is **earlier**?

\( v \) is **not** a descendant, and **not** an ancestor
APPLICATION OF DFS:
STRONG CONNECTEDNESS
Testing existence of all-to-all paths
STRONG CONNECTEDNESS

- In a directed graph,
  - \( v \) is reachable from \( w \) if there is a path from \( w \) to \( v \)
  - we denote such a path \( w \rightarrow v \)
  - A graph \( G \) is strongly connected iff every node is reachable from every other node
  - More formally: \( \forall_{w,v} \exists w \rightarrow v \)
STRONG CONNECTEDNESS

- Is this graph **strongly connected**?
  - a ➔ b ➔ c ➔ d ➔ a
  - No path from c to other nodes.

- How about this one?
  - a ➔ b ➔ c ➔ d ➔ a
  - Yes. One big cycle.
STRONG CONNECTEDNESS

• How about this graph?

Yes. Multiple intersecting cycles.

• How about this one?

No. Two cycles with only a one-directional path between them.
OTHER APPLICATIONS OF CHECKING STRONG CONNECTEDNESS

- You gain some symmetry from knowing a graph is strongly connected
- For example, you can start a graph traversal at any node, and know the traversal will reach every node
- Without strong connectedness, if you want to run a graph traversal that reaches every node in a single pass, you would have to do additional processing to determine an appropriate starting node
OTHER APPLICATIONS OF CHECKING STRONG CONNECTEDNESS

- Useful as a sanity check!
- Suppose you want to run an algorithm that requires strong connectedness, and you believe your input graph is strongly connected
- **Validate** your input by **testing** whether this is true!
- Subtle, difficult-to-detect bugs often result if such an algorithm is run only on one component of a graph
- [More concrete applications once we generalize and talk about strongly connected components…]
A USEFUL LEMMA

- Lemma: a graph is strongly connected iff for any node $s$,
- all nodes are reachable from $s$, and $s$ is reachable from all nodes

Prove both directions:

$(\Rightarrow)$ Suppose for all $u, v$ we have $u \xrightarrow{} v$. Fix any $s$. Node $s$ is reachable from all nodes, and vice versa.

$(\Leftarrow)$ Suppose $s$ is reachable from all nodes and vice versa. For any $u, v$, we have $u \xrightarrow{} s \xrightarrow{} v$, and $v \xrightarrow{} s \xrightarrow{} u$. 
CREATING AN ALGORITHM

- How to use DFS to determine whether every node is reachable from a given node \( s \)?
- How to use DFS to determine whether \( s \) is reachable from every node?

What if we first reverse the direction of every edge? Then \( s \rightarrow v \) in this new graph IFF \( v \rightarrow s \) in the original graph.
THE ALGORITHM

- $IsStronglyConnected(G = \{V, E\})$ where $V = v_1, v_2, \ldots, v_n$
  - $(\text{colour}, d, f) := DFSVisit(v_1, G)$
  - for $i := 1..n$
    - if $\text{colour}[v_i] \neq \text{black}$ then return $false$
  - Construct graph $H$ by \textbf{reversing} all edges in $G$
  - $(\text{colour}, d, f) := DFSVisit(v_1, H)$
  - for $i := 1..n$
    - if $\text{colour}[v_i] \neq \text{black}$ then return $false$
  - return $true$
Every node is black. Next step!

$DFSVisit(a)$ in $G$

(a is arbitrary)
EXAMPLE EXECUTION 1

Every node is black. Next step!

construct graph $H$

$DFSVisit(a)$ in $G$
($a$ is arbitrary)

$DFSVisit(a)$ in $H$

Every node is black. So $G$ is strongly connected!
Every node is black. Next step!

Could the result change if we started at a different node?

construct graph $H$

$DFSVisit(a)$ in $G$
($a$ is arbitrary)

$DFSVisit(a)$ in $H$

Some nodes are not black

So $G$ is not strongly connected!
REVERSING EDGES: ADJACENCY MATRIX

reverse all edges
**REVERSING EDGES: ADJACENCY MATRIX**

**Diagram:**
- Nodes: a, b, c, d, e, f, g
- Edges:
  - a to b
  - b to c
  - c to d
  - d to e
  - e to f
  - f to g

**Adjacency Matrix:**

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
<th>g</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>b</td>
<td>1</td>
<td></td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>c</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>d</td>
<td>1</td>
<td>1</td>
<td></td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>e</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>f</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>g</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Reverse all edges:**
- a to d
- b to e
- c to f
- d to g
- e to c
- f to b
- g to a
REVERSING EDGES: ADJACENCY MATRIX

reverse all edges

source

target
REVERSING EDGES: ADJACENCY MATRIX

reverse all edges

adjacency matrix

source

target

1 1 1 1 1 1 1

1 1 1 1 1 1 1
REVERSING EDGES: ADJACENCY MATRIX

Reverse all edges

source

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
<th>g</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>b</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>c</td>
<td></td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>d</td>
<td></td>
<td></td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>e</td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>f</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>g</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

target

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
<th>g</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>b</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>c</td>
<td></td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>d</td>
<td></td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>e</td>
<td></td>
<td></td>
<td>1</td>
<td></td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>f</td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td></td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>g</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td></td>
<td>1</td>
</tr>
</tbody>
</table>
### REVERSING EDGES: ADJACENCY MATRIX

**Diagram:**
- Original graph with edges from `a` to `b`, `b` to `c`, `c` to `f`, `f` to `e`, `e` to `g`, and `d` to `a`

**Matrix:**

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
<th>g</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>b</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>c</td>
<td></td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>d</td>
<td></td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>e</td>
<td></td>
<td></td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>f</td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>g</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

**Adjacency Matrix:**

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
<th>g</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>b</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>c</td>
<td></td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>d</td>
<td></td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>e</td>
<td></td>
<td></td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>f</td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>g</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

**Reverse all edges:**
- Reverse all the edges in the graph

**Matrix after reversing edges:**

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
<th>g</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>b</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>c</td>
<td></td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>d</td>
<td></td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>e</td>
<td></td>
<td></td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>f</td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>g</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

**New Adjacency Matrix:**

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
<th>g</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>b</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>c</td>
<td></td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>d</td>
<td></td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>e</td>
<td></td>
<td></td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>f</td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>g</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
REVERSING EDGES: ADJACENCY MATRIX

reverse all edges

source

target
REVERSING EDGES: ADJACENCY **MATRIX**

```
  a  b  c  d  e  f  g
a  1
b  1
1  1
c  1
1  1
d  1
e  1
f  1
1  1
g
```

Reverse all edges:

```
  a  b  c  d  e  f  g
a  1
b  1
1  1
c  1
1  1
d  1
e  1
f  1
1  1
g
```
REVERSING EDGES:
ADJACENCY MATRIX

reverse all edges

source

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
<th>g</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>b</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>c</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>d</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>e</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>f</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>g</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

target

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
<th>g</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>b</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>c</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>d</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>e</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>f</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>g</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
**REVERSING EDGES:**

**ADJACENCY MATRIX**

Can do matrix transpose, or can just swap variables for source & target in your code!

Complexity?

---

**Matrix Representation:**

**Source:**

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
<th>g</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>b</td>
<td></td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>c</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>d</td>
<td>1</td>
<td></td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>e</td>
<td></td>
<td>1</td>
<td></td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>f</td>
<td></td>
<td></td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>g</td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

**Target:**

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
<th>g</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>b</td>
<td></td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>c</td>
<td></td>
<td></td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>d</td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>e</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>f</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>g</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
**REVERSING EDGES: ADJACENCY LISTS**

**Complexity?**

```
1 TransposeLists(adj[1..n])
2    newAdj = new array of n lists
3    for u = 1 .. n
4       for v in adj[u]
5           newAdj[v].insert(u)
6    return newAdj
```
RUNTIME COMPLEXITY
FOR ADJACENCY LIST REPRESENTATION?

- IsStronglyConnected($G = \{V, E\}$) where $V = v_1, v_2, \ldots, v_n$
  - $(\text{colour}, d, f) := \text{DFSVisit}(v_1, G)$
  - for $i := 1..n$
    - if colour[$v_i$] ≠ black then return false
  - Construct graph $H$ by reversing all edges in $G$
  - $(\text{colour}, d, f) := \text{DFSVisit}(v_1, H)$
  - for $i := 1..n$
    - if colour[$v_i$] ≠ black then return false
  - return true

$O(n + m)$