CS 341: ALGORITHMS

Lecture 11: 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
(UNDIRECTED) BIPARTITE GRAPHS AND BFS

- 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 $B$.
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?

Edge with both endpoints in $B$!
PROOF
PART 1: ODD CYCLE $\Rightarrow$ NOT BIPARTITE

- Suppose there is an **odd** length cycle $v_1, v_2, \ldots, v_{2k+1}, v_1$

  - And $v_4 \in B$
  - And $v_3 \in R$
  - Then we must have $v_2 \in B$
  - And $v_3 \in R$
  - WLOG let $v_1 \in R$
  - And finally $v_{2k+1} \in R$ !!
  - Both endpoints in $R$! Contradiction!

And so on, alternating...

And $v_4 \in B$
PROOF

PART 2: ALL CYCLES HAVE EVEN LENGTH \( \Rightarrow \) 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

\( G \)

\[ G = \begin{array}{c}
\text{node} \\
\text{distance}
\end{array} \]

\begin{align*}
\text{node} & \quad \text{distance} \\
v_i & \quad 0 \\
8 & \quad 1 \\
3 & \quad 1 \\
5 & \quad 2 \\
6 & \quad 2 \\
1 & \quad 2 \\
2 & \quad 3 \\
4 & \quad 3 \\
\end{align*}

\( R = \textbf{odd} \ d(v) \)
\( B = \textbf{even} \ d(v) \)

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 \(d(u) = \text{length of shortest path} v_i \to \cdots \to u\)

- \(d(u) = \text{odd}\)
- \(d(v) = \text{odd}\)

\[\cdots \]

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

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!

So there is no edge \((u, v)\) where \(u, v \in R\) (case \(B\) is similar)
Algorithm for Testing Bipartiteness

1. Bipartition(adj[1..n])
2.   colour[1..n] = [white, ..., white]
3.   dist[1..n] = [infty, ..., infty]
4.   for start = 1..n
5.     if colour[start] is white
6.        BFS(adj, start, colour, dist)
7.   for edge in adj
8.     let u and v be endpoints of edge
9.     if (dist[u]%2) == (dist[v]%2) then
10.        return NotBipartite
11. B = nodes u with even dist[u]
12. R = nodes u with odd dist[u]
13. 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 $O(n + m)$
DEPTH FIRST SEARCH

Bread first search

Depp first search

DEPTH FIRST SEARCH
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

global variables:
  pred[1..n] = [null, null, ..., null]
  colour[1..n] = [white, white, ..., white]
  d[1..n] = [0, 0, ..., 0] // discovery times
  f[1..n] = [0, 0, ..., 0] // finish times
  time = 0

DepthFirstSearch(adj[1..n])
  for v = 1..n
    if colour[v] == white
      DFSVisit(v)
  DFSVisit(adj[1..n], v)
    colour[v] = gray
    time = time + 1
    d[v] = time
    for each w in adj[v]
      if colour[w] == white
        pred[w] = v
        DFSVisit(w)
    colour[v] = black
    time = time + 1
    f[v] = time

Example execution
starting at node 1

d[1] = 1
f[1] = 10
d[2] = 2
f[2] = 9
d[3] = 3
f[3] = 4
d[4] = 5
f[4] = 6
d[5] = 7
f[5] = 8
d[6] = 11
f[6] = 12
As in breadth first search, \( \text{pred}[] \) array induces a forest. Let’s match the graph’s edge directions (opposite from pred).

DepthFirstSearch(adj[1..n])
for \( v = 1..n \)
if colour[\( v \)] == white
DFSVisit(\( v \))

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 $DFSVisit(v)$ 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 $DFSVisit(v)$ 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)**

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

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 descendant 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**

---

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

### 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>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:**

\( (v \text{ is discovered during } DFSVisit(u)) \)

\( \iff (v \text{ is white-reachable from } u \text{ when we call } DFSVisit(u)) \)

\( \iff (v \text{ is a descendant of } u \text{ in the DFS forest}) \)

\( \iff (v \text{ turns grey after } u \text{ and black before } u) \)

\( \iff (I_v \text{ nested inside } I_u) \)

---

\( v \) discovered during \( DFSVisit(u) \)

but not directly from \( u \) (or \( \{u, v\} \) would be a tree edge)

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

\( v \) is already discovered!

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

\( v \) is a descendant of \( u \)

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

\( v \) is not a descendant, and not an ancestor

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

That call terminates before \( DFSVisit(u) \) 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.

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

<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>

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

So, \( I_v \) must be earlier.
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>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>black</td>
<td>( d[v] &lt; f[v] &lt; d[u] &lt; f[u] )</td>
</tr>
</tbody>
</table>

Recall: \( (v \text{ is discovered during } DFSVisit(u)) \)

\( \iff (v \text{ is white-reaching from } u \text{ when we call } DFSVisit(u)) \)

\( \iff (v \text{ is a descendant of } u \text{ in the DFS forest}) \)

\( \iff (v \text{ turns grey after } u \text{ and black before } u) \)

\( \iff (I_v \text{ nested inside } I_u) \)

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

So, \( I_v \) must be earlier.

Intervals \( I_u \) and \( I_v \) must be **disjoint**.

But which is **earlier**?

\( v \) is not a descendent, and **not** an ancestor
APPLICATION OF DFS (OR BFS):
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 \( w \xrightarrow{} v \)
  - A graph \( G \) is strongly connected iff every node is reachable from every other node
  - More formally: \( \forall w, v \in G \exists w \xrightarrow{} v \)
STRONG CONNECTEDNESS

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

- How about this one?
  - 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

Proof: ($\Rightarrow$) Suppose $G$ is strongly connected. Then for all $u, v$ we have $u \rightarrow v$. Fix any $s$. Node $s$ is reachable from all nodes, and vice versa.

($\Leftarrow$) Suppose some $s$ is reachable from all nodes and vice versa. For any $u, v$, we have $u \rightarrow s \rightarrow v$, and $v \rightarrow s \rightarrow u$. So $G$ is strongly connected.
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?

DFS from $s$ and see if every node turns black

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

DFS from $s$
THE ALGORITHM

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

Every node is black. Next step!

$DFSVisit(a)$ in $H$

Every node is black. So $G$ is strongly connected!
**EXAMPLE EXECUTION 2**

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)`

Every node is black. Next step!

`DFSVisit(a) in H`

Some nodes are not black

No path from those nodes to `a`

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

\[
\begin{array}{cccccccc}
\text{source} & a & b & c & d & e & f & g \\
\hline
a & & & & & & & 1 \\
b & 1 & & & & & & \\
c & & 1 & & & & & 1 \\
d & 1 & & & & & & \\
e & & & & & & & 1 \\
f & 1 & 1 & & & & & 1 \\
g & & & & & & & \\
\text{target} & a & b & c & d & e & f & g \\
\hline
a & & & & & & & \\
b & & & & & & & \\
c & 1 & & & & & & \\
d & & 1 & & & & & \\
e & 1 & 1 & & & & & \\
f & & & & & & & \\
g & & & & & & & 1 \\
\end{array}
\]

reverse all edges
REVERSING EDGES: ADJACENCY MATRIX

**Source**

1. **a** → **b**
2. **b** → **c**
3. **c** → **d**
4. **d** → **e**
5. **e** → **f**
6. **f** → **g**
7. **g** → **a**

**Target**

1. **a** → **f**
2. **b** → **g**
3. **c** → **e**
4. **d** → **b**
5. **e** → **c**
6. **f** → **d**
7. **g** → **a**

Reverse all edges:

1. **a** → **c**
2. **c** → **b**
3. **b** → **d**
4. **d** → **e**
5. **e** → **f**
6. **f** → **g**
7. **g** → **a**

Number of edges after reversing: 7
REVERSING EDGES: ADJACENCY **MATRIX**

```
<table>
<thead>
<tr>
<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>
</tr>
<tr>
<td>b</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></td>
<td></td>
<td></td>
</tr>
<tr>
<td>d</td>
<td>1</td>
<td></td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>e</td>
<td></td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>f</td>
<td></td>
<td></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>
</tr>
</tbody>
</table>
```

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

reverse all edges
REVERSING EDGES:
ADJACENCY MATRIX

reverse all edges
REVERSING EDGES: ADJACENCY MATRIX

reverse all edges
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></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></td>
<td>1</td>
<td></td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>e</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>f</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>g</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></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>1</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></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>e</td>
<td>1</td>
<td></td>
<td></td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>f</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>g</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>
REVERSING EDGES: ADJACENCY MATRIX

reverse all edges
REVERSING EDGES: ADJACENCY MATRIX

reverse all edges

source


target

43
REVERSING EDGES: ADJACENCY MATRIX

Can do matrix transpose, or can just treat rows as columns and vice versa in your code.

Complexity?

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

\[
M_E
\]

\[
(M_E)^T
\]
REVERSING EDGES: ADJACENCY LISTS

reverse edges

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?

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

\textit{O}(n + m)