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
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 $B$!
PART 1: ODD CYCLE ⇒ NOT BIPARTITE

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

And so on, alternating...

WLOG let $v_1 \in R$

Then we must have $v_2 \in B$

And $v_3 \in R$

And $v_4 \in B$

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

Both endpoints in $R$! Contradiction!

And $v_{2k} \in B$

Until $v_{2k} \in B$

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

$G$

$R = \text{odd } d(v)$

$B = \text{even } d(v)$

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

\[d(u) = \text{odd}\]

\[d(v) = \text{odd}\]

...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

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 $O(n + m)$
DEPTH 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.
Example execution starting at node 1

d[1] = 1

d[2] = 2

d[3] = 3

d[4] = 4

d[5] = 5

d[6] = 6

f[1] = 10
f[2] = 9
f[3] = 4
f[4] = 6
f[5] = 8
f[6] = 12

DFSVisit(1)

DFSVisit(6)

d[1] = 1

d[2] = 2

d[3] = 3

d[4] = 5

d[5] = 7

d[6] = 11

f[1] = 10
f[2] = 9
f[3] = 4
f[4] = 6
f[5] = 8
f[6] = 12

deepth first search algorithm

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
**DFS TREE / FOREST**

- As in breadth first search, `pred[]` array induces a **forest**
- Let’s match the graph’s edge directions (opposite from `pred`)

**Graph**

```
1 → 2 → 4
<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
</table>
\[d[1]=1\] | \[f[2]=9\] |
```

```
2 → 3 → 5
<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
</table>
\[d[2]=2\] | \[f[5]=8\] |
```

```
4 → 6
<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
</table>
```

**DFS forest**

```
1 → 2 → 4
<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
</table>
\[d[1]=1\] | \[f[2]=9\] |
```

```
2 → 3 → 5
<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
</table>
\[d[2]=2\] | \[f[5]=8\] |
```

```
4 → 6
<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
</table>
```

**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)

Home exercise: complexity with adjacency matrix?

```
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
```

- 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**

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

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

**iff** \( v \) is **white-reachable** from \( u \) when \( DFSVisit(u) \) is called
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 \( u \))
- \( (\text{discovery/finish time interval } I_v \text{ is nested inside } I_u) \)
- \( (v \) 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: \((v\ \text{is discovered during}\ DFSVisit(u))\)

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

\(v\) 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 a child of \(u\)

\(v\) is already discovered!

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

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

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

\[
\text{... by another recursive call that } DFSVisit(u) \text{ makes when it inspect a previous edge} \\
\text{That call terminates before } DFSVisit(u) \text{ inspects } \{u, v\} \\
\text{And it colors } v \text{ black!}
\]
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.
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>Q8?</td>
<td>Q7?</td>
</tr>
</tbody>
</table>

Recall: \( v \) is discovered during \( DFSVisit(u) \)

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

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

\[ \Rightarrow (v \text{ turns grey after } u \text{ and black before } u) \]

\[ \Rightarrow (I_v \text{ nested inside } I_u) \]

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.

\( v \) is not a descendant, and not an ancestor.
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:** \( v \) is discovered during \( DFSVisit(u) \)
\[ \Leftrightarrow (v \text{ is white-reaching from } u \text{ when we call } DFSVisit(u)) \]
\[ \Leftrightarrow (v \text{ is a descendant of } u \text{ in the DFS forest}) \]
\[ \Leftrightarrow (v \text{ turns grey after } u \text{ and black before } u) \]
\[ \Leftrightarrow (I_v \text{ nested inside } I_u) \]

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 \( w \rightarrow v \)
• A graph \( G \) is strongly connected iff every node is reachable from every other node
• More formally: \( \forall_{w,v} \in \mathcal{V} \exists w \rightarrow v \)
STRONG CONNECTEDNESS

• Is this graph *strongly connected*?

![Graph 1](image1)

No path from c to other nodes.

• How about this one?

![Graph 2](image2)

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 \( u \leadsto v \). 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 \( u \leadsto s \leadsto v \), and \( v \leadsto s \leadsto u \). So \( G \) is strongly conn.
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 $colour[v_i] \neq \text{black}$ then return $false$
  • Construct graph $H$ by reversing all edges in $G$
  • $(colour, d, f) := DFSVisit(v_1, H)$
  • for $i := 1..n$
    • if $colour[v_i] \neq \text{black}$ then return $false$
  • return $true$

How?
EXAMPLE EXECUTION 1

Every node is black. Next step!

DFSVisit(a) in G
(a is arbitrary)
EXAMPLE EXECUTION 1

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

Every node is black. Next step!

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

Some nodes are not black

No path from those nodes to a

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

reverse all edges
REVERSING EDGES: ADJACENCY MATRIX

reverse all edges
### Reversing Edges: Adjacency Matrix

#### 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></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>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>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></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>1</td>
<td>1</td>
</tr>
</tbody>
</table>

**Reverse all edges**: The diagram shows the original graph with arrows indicating the directions of the edges. The text explains the process of reversing all edges. The adjacency matrix is shown for both the source and target graphs, with the values indicating the presence or absence of edges between nodes. The reversal process is visually represented in the diagram.
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></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></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>c</td>
<td></td>
<td>1</td>
<td></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>1</td>
<td></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>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></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>b</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>c</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>d</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>e</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>f</td>
<td></td>
<td></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>
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>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>c</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>d</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>e</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>f</td>
<td></td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>g</td>
<td></td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></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></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>c</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>d</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>e</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>f</td>
<td></td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>g</td>
<td></td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
REVERSING EDGES: ADJACENCY MATRIX

Reverse all edges
REVERSING EDGES:
ADJACENCY MATRIX

reverse all edges
REVERSING EDGES: ADJACENCY MATRIX

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

Complexity?

reverse all edges

Complexity?
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
```
• 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 \(\text{colour}[v_i] \neq \text{black}\) then return \(\text{false}\)
  • Construct graph \(H\) by \textbf{reversing} all edges in \(G\)
  • \((\text{colour}, d, f) := \text{DFSVisit}(v_1, H)\)
  • for \(i := 1..n\)
    • if \(\text{colour}[v_i] \neq \text{black}\) then return \(\text{false}\)
  • return \(\text{true}\)