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

Both endpoints in $R$! Contradiction!

WLOG let $v_1 \in R$

Then we must have $v_2 \in B$

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

What happens if I create an odd length cycle?

PART 2: ALL CYCLES HAVE EVEN LENGTH ⇒ BIPARTITE

Let $v$ be any node, and $d(v)$ be the distance from $v$ to $v$

Partition nodes by even vs odd distances

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

\[d(u) = odd\]
\[d(v) = odd\]

Recall the combined path \(v_i \rightarrow \cdots \rightarrow v_j \rightarrow v_k \rightarrow v_i\) 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. Call BFS on each component to calculate distances for each node
2. Modified BFS that reuses the same colour array and same dist array
3. If both even or both odd return Actual Bipartition
4. Run time complexity ?

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

DEEP FIRST SEARCH ALGORITHM

DFS TREE / FOREST

- As in breath first search, \(pred[]\) array induces a tree
- Let’s match the graph's edge directions (opposite from pred)

Example execution starting at node 1
BASIC DFS PROPERTIES TO REMEMBER

- Nodes start white. Also gets a discovery time $d[v]$ at this point.
- A node $v$ turns gray when it is discovered, which is when the first call to $DFS\text{Visit}(v)$ happens. Also gets a finish time $f[v]$ at this point.
- 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 $DFS\text{Visit}(v)$ does, so the neighbours of $v$ turn black before $v$.

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.

EXPLORING $D[]$, $F[]$ AND COLOUR[]

- Observe: every node $v$ that is white-reachable from $u$ when we first call $DFS\text{Visit}(u)$ becomes gray after $u$ and black before $u$ (so $I_v$ is nested inside $I_u$).

RUNTIME COMPLEXITY OF DFS (FOR ADJ. LISTS)

Home exercise: complexity with adjacency matrix!

DEFINITIONS

- Definition: we use $I_v$ to denote $(d[v], f[v])$, which we call the interval of $v$.
- Definition: $v$ is white-reachable from $u$ if there is a path from $u$ to $v$ containing only white nodes (excluding $u$).

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 $DFS\text{Visit}(u))$
  - $(v$ turns grey after $u$ and black before $u)$
  - $(v$ is discovered and $u$ is nested inside $I_v)$
  - $(v$ is discovered during $DFS\text{Visit}(u))$
  - $(v$ is a descendent 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>white</td>
<td>(t[u] &lt; d[u] &lt; f[u])</td>
</tr>
<tr>
<td>back</td>
<td>white</td>
<td>(d[u] &lt; t[v] &lt; f[v])</td>
</tr>
<tr>
<td>cross</td>
<td>grey</td>
<td>(d[v] &lt; t[u] &lt; f[u])</td>
</tr>
</tbody>
</table>

Recall:
- \((v)\) is discovered during DFSVisit \((u)\)
- \(v\) is white-reachable from \(u\) when we call DFSVisit \((u)\)
- \(v\) is a descendant of \(u\) in the DFS forest
- \(v\) is not a descendant and not an ancestor

**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 \(f[u] \in I_v\)
  - WLOG suppose \(d[v] \in I_u\)
  - Then \(v\) is discovered during DFSVisit \((u)\)
  - So, \(v\) must turn grey after \(u\) and black before \(u\)
  - \(\Rightarrow f[v] < f[u]\)
  - So the intervals are nested. QED

**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** if
  - every node is reachable from every other node
  - More formally: \(V \rightarrow W \rightarrow V\)

**STRONG CONNECTEDNESS**

- Is this graph **strongly connected**?
- How about this one?

No path from \(c\) to other nodes.
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 nodes $x$, all nodes are reachable from $x$, and $x$ is reachable from all nodes.

CREATE 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?

THE ALGORITHM

- $I_1$ isStronglyConnected($G = (V,E)$) where $V = v_1, v_2, ..., v_n$
  - $(\alpha, \delta) = DFSVisit(v_i, G)$
  - $t = 1..n$
    - if $\alpha[v_t] = \text{black}$ then return false
  - Construct graph $H$ by reversing all edges in $G$
  - $(\alpha, \delta) = DFSVisit(v_i, H)$
  - $t = 1..n$
    - if $\alpha[v_t] = \text{black}$ then return false
    - return true

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

DFS from $s$ and reverse the direction of every edge

Then run DFS in this new graph and see if every node turns black
Every node is black.

Next step!

\( \text{DFSVisit}(a) \) in \( G \) (\( a \) is arbitrary)

Exercise 1:

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

Exercise 2:

Some nodes are not black.

No path from those nodes to \( a \).

So \( G \) is not strongly connected!

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

Reverse all edges.

Exercise 3:

Reverse all edges.

Exercise 4:

Reverse all edges.

Exercise 5:

Reverse all edges.
Can do matrix transpose, or just swap variables for source & target in your code!

Complexity? $O(n^2)$

Reverse all edges

Complexity?

Reverse edges

Complexity?

Runtime Complexity for Adjacency List Representation:

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

$O(n + m)$