(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 \( B \).

\[ G \]

\[ R \]

\[ B \]

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?

\[ G \]

\[ R \]

\[ B \]

PROOF

PART 1: ODD CYCLE \( \Rightarrow \) NOT BIPARTITE

- Suppose there is an odd length cycle \( v_1, v_2, \ldots, v_{2k+1}, v_1 \) until \( v_{2k} \in B \) and finally \( v_{2k+1} \in R \).

And so on, alternating...

\[ G \]

\[ R \]

\[ B \]

WLOG let \( v_1 \in R \)

And finally \( v_{2k+1} \in R \) II Both endpoints in \( R \) Contradiction!

PROOF

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

- Let \( v \) be any node, and \( d(v) \) be the distance from \( v \) to \( v \).

- Partition nodes by even vs odd distances.

\[ G \]

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

\[ B = \text{even } d(v) \]
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 \(d(u) = \text{length of shortest path } v_i \rightarrow \cdots \rightarrow u\)

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

Recall \(d(v) = \text{length of shortest path } v_i \rightarrow \cdots \rightarrow v\)

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

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

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
4. Return an actual bipartition
5. Runtime complexity?
6. Can be done in \(O(n + m)\)

DEPTH FIRST SEARCH

A depth-first search uses a stack (or recursion) instead of a queue.
We define predecessors and colour values 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

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

DFS TREE / FOREST

• Each top level DFSVisit calls the root of a tree
• DFS forest
**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

**Definition:** If \( v \) turns gray before \( u \) inside Recursive calls on neighbours end, then: \( v \) is a descendent of \( u \) in the DFS forest

**Theorem:** Let \( u \) be any node. The following statements are all equivalent:
- \( v \) **is white-reachable** from \( u \) when we first call \( \text{DFSVisit}(u) \)
- \( v \) **is a descendant** of \( u \) in the DFS forest
- \( v \) **turns gray** after \( u \) and **black** before \( u \)
- \( v \) is discovered during \( \text{DFSVisit}(u) \)
- \( v \) is a **descendant** of \( u \) in the DFS forest

**DEFINITIONS**

- **Definition:** \( I_u \) 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 \( u \))

**RUNTIME COMPLEXITY OF DFS (FOR ADJ. LISTS)**

- **Home exercise:** complexity with adjacency matrix?
- **Total** (forest) mentions over all recursive calls. Total \( \mathcal{O}(n + m) \) runtime!
**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>(d[u] &lt; d[v] &lt; f[v] &lt; f[u])</td>
</tr>
<tr>
<td>back</td>
<td>black</td>
<td>(d[v] &lt; d[u] &lt; f[u] &lt; f[v])</td>
</tr>
<tr>
<td>front</td>
<td>gray</td>
<td>(d[u] &lt; d[v] &lt; f[u] &lt; f[v])</td>
</tr>
<tr>
<td>cross</td>
<td>gray</td>
<td>(d[u] &lt; d[v] &lt; f[v] &lt; f[u])</td>
</tr>
</tbody>
</table>

Recall: (is discovered during DFSVisit(u))
- \(v\) is a child of \(u\) in the DFS tree
- \(v\) is a descendant of \(u\)
- \(v\) is a (parent/ancestor) of \(u\)
- \(v\) is not a descendant (and not an ancestor)

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

**USEFUL FACT: PARENTHESES 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 \(d[u] \in I_v\) or \(f[u] \in I_v\)
  - WLOG suppose \(d[u] \in I_v\)
  - Then \(v\) is discovered during DFSVisit(u)
  - So \(v\) must turn gray after \(w\) and black before \(u\)
  - So \(|v| < |w|\)
  - So the intervals are nested. QED

**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 \(\forall x, y \in V\) \(\exists w \in V\) such that \(w \rightarrow x \rightarrow y\)
STRONG CONNECTEDNESS

• Is this graph strongly connected?
  No path from u to other nodes.

• How about this one?
  Yes. One big cycle.

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.

A USEFUL LEMMA

• Lemma: a graph is strongly connected if for any node s, all nodes are reachable from s, and s is reachable from all nodes.

  Proof: (⇒) Suppose G is strongly connected. Then for all nodes, there is a path to every other node. Fix any s. Node s is reachable from all nodes, and vice versa.

  (⇐) Suppose some s is reachable from all nodes and vice versa. For any s, s has a path to every other node, and vice versa. 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 x and we’ll have if every node turns black.

  What if we first reverse the direction of every edge?

  DFS from y
THE ALGORITHM

\[ IsStrONGLYConnected(G = (V,E)) \] where \( V = v_1, v_2, \ldots, v_n \)

\[ (\text{colour}, d, f) = DFSVisit(v_1, G) \]

for \( i = 1 \) to \( n \)

\[ \text{if \ colour(v_i) \neq black \ then \ return \ false} \]

Construct graph \( H \) by **reversing** all edges in \( G \)

\[ (\text{colour}, d, f) = DFSVisit(v_1, H) \]

for \( i = 1 \) to \( n \)

\[ \text{if \ colour(v_i) \neq black \ then \ return \ false} \]

return true

EXAMPLE EXECUTION 1

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

```
DFSVisit(a) in G  (a is arbitrary)
Every node is black. Next step!
```

```
DFSVisit(a) in H
Every node is black. Next step!
```

REVERSING EDGES:

```
Source | Target
--- | ---
1 | 2
2 | 1
3 | 4
4 | 3
```

reverse all edges

```
Source | Target
--- | ---
1 | 2
2 | 1
3 | 4
4 | 3
```

REVERSING EDGES:

```
Source | Target
--- | ---
1 | 2
2 | 1
3 | 4
4 | 3
```

```
Source | Target
--- | ---
1 | 2
2 | 1
3 | 4
4 | 3
```

reverse all edges
REVERSING EDGES: ADJACENCY MATRIX

\[
\begin{array}{cccc}
\text{source} & a & b & c \\
\text{target} & 1 & 1 & 1 \\
\end{array}
\]

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

\[
(M_E)^T
\]

REVERSING EDGES: ADJACENCY LISTS

\[
\begin{array}{cccc}
\text{source} & a & b & c \\
\text{target} & 1 & 1 & 1 \\
\end{array}
\]

Complexity?

\[
O(n + m)
\]

REVERSING EDGES: ADJACENCY MATRIX

\[
\begin{array}{cccc}
\text{source} & a & b & c \\
\text{target} & 1 & 1 & 1 \\
\end{array}
\]

Complexity?

\[
0(n + m)
\]

RUNTIME COMPLEXITY FOR ADJACENCY LIST REPRESENTATION?

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